

History of Python:

- It was developed by Guido Van Rossum in the late 80's and early 90's at National Research Institute for mathematics and computer science in the Netherlands.
- It is a successor to ABC language.
- Its version 0.9.0 was released in 1991 with features functions, data types.
- Python version 1.0 was released in 1994 with functional programming tools such as lambda, map, and reduce.
- 2.0 released in 2000 with features List comprehension, Unicode and garbage collector.
- 3.0 released in 2008 by removing duplicate programming constructs and modules.
- Latest python version is 3.7.4 (as on 15/8/2019)

Note:

- Surprisingly Python is older than Java, java script and R.
- Why is it called **Python**? When he began implementing **Python**, Guido van Rossum was also reading the published scripts from "Monty **Python's** Flying Circus", a BBC comedy series from the 1970s. Van Rossum thought he needed a **name** that was short, unique, and slightly mysterious, so he decided to call the language **Python**.

Python Features:

Python provides lots of features that are listed below:

1) Easy to Learn and Use

Python is easy to learn and use. It is developer-friendly and high level programming language.

2) Expressive Language

Python language is more expressive means that it is more understandable and readable.

3) Interpreted Language

Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.

4) Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc. So, we can say that Python is a portable language.

5) Free and Open Source

Python language is freely available at [official web address](#). The source-code is also available. Therefore it is open source.

6) Object-Oriented Language

Python supports object oriented language and concepts of classes and objects come into existence.

7) Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our python code.

8) Large Standard Library

Python has a large and broad library and provides rich set of module and functions for rapid application development.

9) GUI Programming Support

Graphical user interfaces can be developed using Python.

10) Integrated

It can be easily integrated with languages like C, C++, JAVA etc.

Literal Constants:

The Literal Constant can be 10, 10.5, 'A' and "hello". These can be directly used in the program. Constant is a fixed value that does not change. Let us see Number and string constants in C.

Numbers:

It is a numerical value. We can use four types' numbers in python. They are Binary numbers, Integral numbers, floating point numbers and complex numbers.

Binary Numbers made up of with 0's and 1's. In python every Binary number begins with either 0b or 0B (zero b). Eg: 0b1110, 0B1111

Integral numbers can be whole numbers or integer number or octal number or Hexadecimal Number.

- Integer numbers** like 10, 34567, 123456778889999907788, -9876. Integer can be small, long, positive or negative. **In python integer numbers are identified by word int.** Integer number never have a decimal point.
- Octal numbers** start with either 0o or 0O (zero capital O). Octal number is made of digits {0-7}. Eg: 0o57 is a valid octal number, 0o876 is not a valid octal number. **Octal numbers identified by oct word.**
- Hexa decimal numbers** start with either 0x or 0X. It is made of digits {0-9} and letters {A-Z or a-z}. Eg: 0xface, 0Xheaf, 0x975b are valid but 0xbeer, 0x5vce are not valid. **Hexa decimal numbers identified by hex word.**

Floating point numbers are like 5.678, 91.45e-2, 91.4E-2 (it equals to 91.4×10^{-2}).

In python Floating numbers are identified by word float which has a decimal point.

Complex numbers have real ,imaginary part like $a+bj$

Eg: $1+5j$, $7-8j$

Complex numbers are identified by the word by **complex**.

Note: Commas are not allowed in numerical. Eg: 3,567 -8,90876 are invalid numbers.

- No size limit for integer number that can be represented in python but floating point number have a range of 10^{-323} to 10^{308} with 15 digits precision.
- Issues with floating point numbers:
 - Arithmetic Overflow Problem:** It occurs when a result is too large in size to be represented. Eg: $2.7e200 * 4.3e200$ you will get infinity.
 - Arithmetic Underflow Problem:** It occurs when a result is too small in size to be represented. Eg: $2.7e-200 / 4.3e200$ you will get infinity.
 - Loss of precision:** Any floating number has a fixed range and precision, the result is just approximation of actual or true value. Slight loss in accuracy is not a concern in practically but in scientific computing it is an issue.

Strings

A *string* is a group of characters.

- **Using Single Quotes (')**: For example, a string can be written as 'HELLO'.
- **Using Double Quotes (")**: Strings in double quotes are exactly same as those in single quotes. Therefore, 'HELLO' is same as "HELLO".
- **Using Triple Quotes (''' ''')**: You can specify multi-line strings using triple quotes. You can use as many single quotes and double quotes as you want in a string within triple quotes.

Examples:

<pre>>>> 'Hello' 'Hello'</pre>	<pre>>>> "HELLO" 'HELLO'</pre>	<pre>>>> '''HELLO''' 'HELLO'</pre>
---	---	---

Escape Sequences

Some characters (like " , \) cannot be directly included in a string. Such characters must be escaped by placing a backslash before them.

Example:

```
>>> print("The boy replies, \"My name is Aaditya.\")  
The boy replies, "My name is Aaditya."
```

Escape Sequence	Purpose	Example	Output
\\	Prints Backslash	print("\\")	\
\'	Prints single-quote	print("\'")	'
\"	Prints double-quote	print("\"")	"
\a	Rings bell	print("\a")	bell rings
\f	Prints form feed character	print("Hello\fWorld")	Hello World
\n	Prints newline character	print("Hello\nWorld")	Hello World
\t	Prints a tab	print("Hello\tWorld")	Hello World
\o	Prints octal value	print("\o58")	8
\x	Prints hex value	print("\xa7")	A

Example 2: `print("Welcome to \\Vmeg\\")`

Output: Welcome to \Vmeg\

Example 3: `print("Welcome to \'Vmeg'\")`

Output: Welcome to 'Vmeg'

Example 4: `print("Welcome to\b Vmeg ")`

Output: Welcom to Vmeg

Example 5: `print("Welcome to\r Vmeg")`

Output:

Use of Triple Quotes:

The triple quotes can be single or double i.e. `'''` or `"""`

- 1) Triple quotes can be used for multi line string literals

```
Example: print(""" Hello  
                How are  
                You?""")
```

Output: Hello

```

How are
You?
Example: print("""Hello
                How are
                You?""")
Output:Hello
        How are
        You?

```

- 2) Triple quotes can be used for printing a single quotes or double quotes

```

Example: print(' Hello
                How are'
                You?')
Output:Hello
        How are'
        You?

```

```

Example: print("Hello
                How are"
                You?")
Output:Hello
        How are"
        You?

```

String Formatting:

The format function can be used to control the display of strings.

Format(string, format specifier)

The strings can be left, right, central justification in a specified width.

Ex: format("HELLO", "<10") -left justification

o/p:

H	E	L	L	O					
---	---	---	---	---	--	--	--	--	--

Rightt justification(>)

Ex: format("HELLO", ">10")

					H	E	L	L	O
--	--	--	--	--	---	---	---	---	---

Central Justification (^)


```
Ex:format("HELLO","^10")
```

		H	E	L	L	O			
--	--	---	---	---	---	---	--	--	--

By using format function, we can fill the width with other characters also

```
Ex:print("Hello",format("*","<10"))
```

o/p: hello *****

Identifiers:

An identifier is a name used to identify a variable, function, class, module, or object.

In **Python**, an identifier is like a noun in **English**.

Identifier helps in differentiating one entity from the other. For example, name and age which speak of two different aspects are called identifiers.

Python is a case-sensitive programming language. Means, Age and age are two different **identifiers** in **Python**.

Let us consider an example:

```
Name = "VMEG"
```

```
name = "VMEG"
```

Here the identifiers Name and name are different, because of the difference in their case.

Below are the rules for writing identifiers in Python:

1. Identifiers can be a combination of lowercase letters (a to z) or uppercase letters (A to Z) or digits (0 to 9) or an underscore (_).

Eg: myClass, var_1, print_this_to_screen, _number are valid **Python** identifiers.

2. An identifier can't start with a digit.

Eg: 1_variable is invalid, but variable_1 is perfectly fine.

3. **Keywords** cannot be used as identifiers. (**Keywords** are reserved words in **Python** which have a special meaning).

Eg: Some of the **keywords** are def, and, not, for, while, if, else and so on.

4. **Special symbols** like !, @, #, \$, % ,space or gap etc. are not allowed in identifiers. Only one special symbol underscore (_) is allowed.

Eg: company#name, \$name, email@id ,simple interest are **invalid Python** identifiers.

5. **Identifiers** can be of any length.

Variables:

Variable is a name used to store a constant. Like other languages like c, c++, java, no need to declare or write type of a variable.

You can store any piece of information in a variable. Variables are nothing but just parts of your computer's memory where information is stored. *To be identified easily, each variable is given an appropriate name.*

Examples of valid variable names are a, b, c, sum, __my_var, num1, r, var_20, first, etc.

Examples of invalid variable names are 1num, my-var, %check, Basic Sal, H#R&A, etc.

UNDERSTANDING VARIABLES:

In **Python** a variable is a reserved memory location used to **store** values.

For example in the below code snippet, age and city are variables which store their respective values.

```
age = 21  
city = "Tokyo"
```

Usually in programming languages like **C**, **C++** and **Java**, we need to declare **variables** along with their **types** before using them. **Python** being a dynamically typed language, there is no need to declare variables or declare their types before using them.

Python has no command for declaring a variable. A variable is created the moment, a value is assigned to it.

The **equal-to (=)** operator is used to assign value to a variable.

Note: Operators are special **symbols** used in programming languages that represent particular actions. = is called the **assignment operator**.

For example :

`marks = 100` # Here marks is the variable and 100 is the value assigned to it.

Note: As a good programming practice, we should use meaningful names for variables. For example, if we want to store the **age** of a person, we should not name the variable as `x`, or `y` or `a`, or `i`. Instead we should name the variable as `age`.

Python interpreter automatically determines the type of the data based on the data assigned to it.

Note: Henceforth, where ever we say Python does this or that, it should be understood that it is the Python interpreter (the execution engine) which we are referring to.

In Python, we can even change the type of the variable after they have been set once (or initialized with some other type). This can be done by assigning a value of different type to the variable.

A variable in **Python** can hold any type of value like 12 (integer), 560.09 (float), "Vmeg" (string) etc.

Associating a **value** with a **variable** using the assignment operator (`=`) is called as Binding.

In **Python**, **assignment** create references, not copies.

We cannot use **Python** variables without assigning any value.

If we try to use a variable without assigning any value then, Python interpreter shows **error** like **"name is not defined"**.

Let us assign the variables `value1` with **99**, `value2` with **"Hello Python"** and `value3` with **"Hello World"**, and then print the result.

Sample Input and Output:

99

Hello Python

Hello World


```
#python program
Value1=99
Value2="Hello Python"
Value3="Hello World"
print(Value1)
print(Value2)
print(Value3)
```

ASSIGNMENT OF VARIABLES:

Python variables do not need explicit declaration to reserve memory space.

The declaration happens automatically when you assign a value to a variable.

The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand (or expression) to the right of the = operator is the value stored in the variable.

Let us consider the below example:

```
counter = 100 # An integer assignment
print(counter) # Prints 100
miles = 1000.50 # A floating point assignment
print(miles) # Prints 1000.5
name = "John" # A string assignment
print(name) # Prints John
```

In the above example **100**, **1000.50**, and **"John"** are the values assigned to the variables **counter**, **miles**, and **name** respectively.

The above example program produces the result as:

```
100
1000.5
```

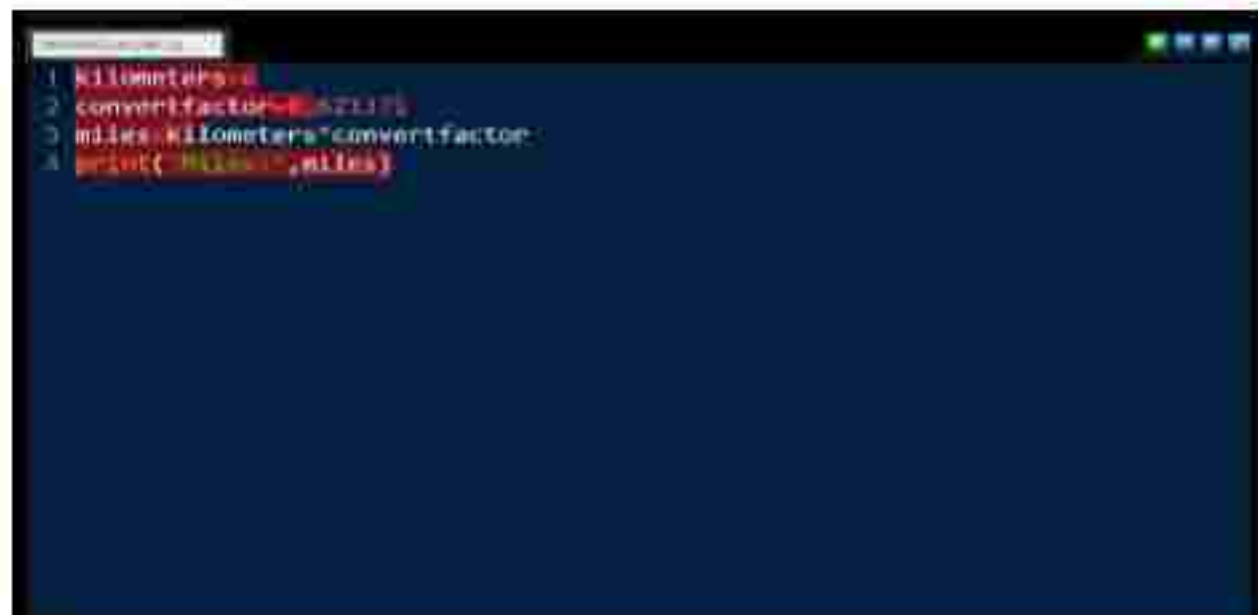
John

Write a program which does the following:

1. Create a variable kilometers and assign 6 to it.
2. Create a variable convertfactor and assign 0.621371 to it.
3. Calculate the product of kilometers and convertfactor and assign it to a variable miles
4. Print the values of miles as "Miles:" miles

Expected Output:

Miles:-3.7282260000000003

A screenshot of a code editor with a dark blue background. The code is written in a light blue font. It consists of four lines: 1. kilometers=6, 2. convertfactor=0.621371, 3. miles=kilometers*convertfactor, 4. print("Miles:",miles). The code is enclosed in a light blue border.

```
1 kilometers=6
2 convertfactor=0.621371
3 miles=kilometers*convertfactor
4 print("Miles:",miles)
```

UNDERSTANDING MULTIPLE ASSIGNMENTS

Python allows you to assign a single value to several variables **simultaneously**.

Let us consider an example:

```
number1 = number2 = number3 = 100
```

Here an **integer object** is created with the value 100, and all the three variables are references to the same memory location. This is called chained assignment.

We can also assign multiple objects to multiple variables, this is called multiple assignment.

Let us consider the below example:

```
value1, value2, value3 = 1, 2.5, "Ram"
```

Here the **integer object** with value 1 is assigned to variable **value1**, the **float object** with value 2.5 is assigned to variable **value2** and the **string object** with the value "Ram" is assigned to the variable **value3**.

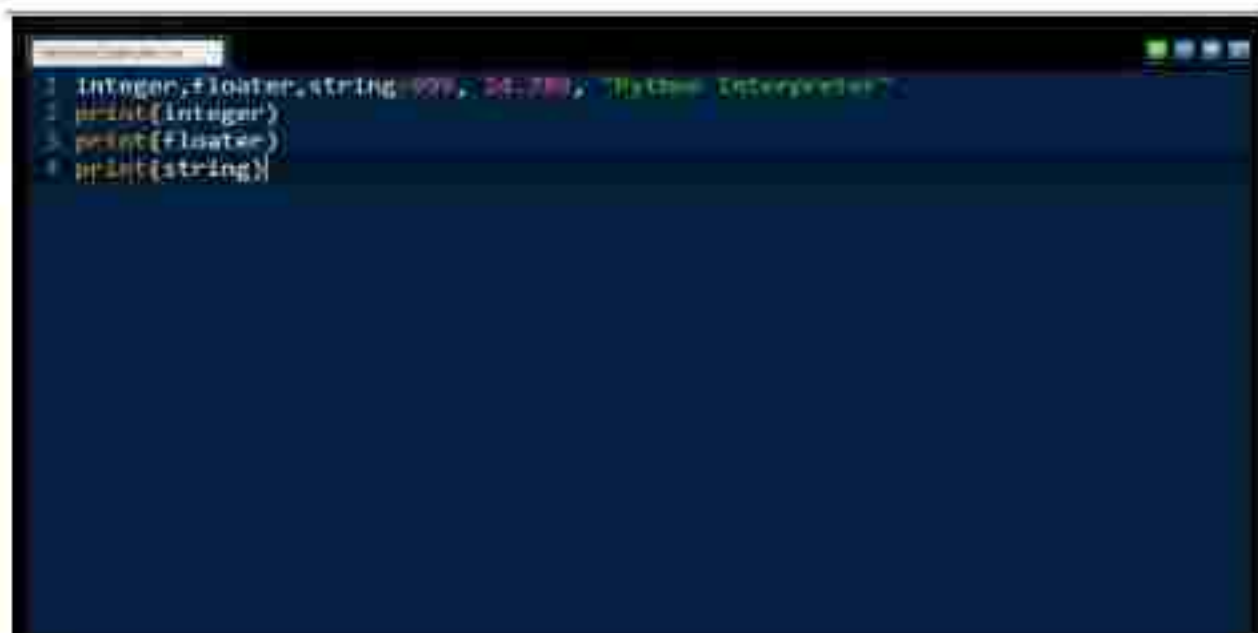
Write a program to assign the integer **999**, float **24.789** and string **"Python Interpreter"** to three variables using multiple assignment and print them individually.

Sample Input and Output:

999

24.789

Python Interpreter



```
1 integer, floater, string = 999, 24.789, "Python Interpreter"
2 print(integer)
3 print(floater)
4 print(string)
```

CHAINED ASSIGNMENT:

In **Python**, assignment statements do not return a value. Chained assignment is recognised and supported as a special case of the assignment statement.

A statement like `a = b = c = x` is called a chained assignment in which the value of `x` is assigned to multiple variables `a`, `b`, and `c`.

Let us consider a simple example:

```
a = b = c = d = 10
print(a) # will print result as 10
print(b) # will print result as 10
print(c) # will print result as 10
print(d) # will print result as 10
```

Here, we initialising the `a`, `b`, `c`, `d` variables with value 10.

Write a program to assign a user given value to `a`, `b`, `c` variables.

At the time of execution, the program should print message on the console as:

Enter a value:

For example, if the user gives the input as:

Enter a value: 100

then the program's result is as follows :

```
Value of a: 100
Value of b: 100
Value of c: 100
```

Note: Let us assume that `input()` is used to read values given by the user. We will learn about `input()` later sections.

Here,

```
a = b = c = str
```

Sample Input and Output:

```
Enter a value: Hello Python
Value of a: Hello Python
Value of b: Hello Python
Value of c: Hello Python
```

[illegible]

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. [Numbers](#)
2. [String](#)
3. [List](#)
4. [Tuple](#)
5. [Dictionary](#)

Python Keywords

Python Keywords are special reserved words which convey a special meaning to the compiler/interpreter. Each keyword have a special meaning and a specific operation. These keywords can't be used as variable. Following is the List of Python Keywords.

True	False	None	and	as
Assert	def	Class	continue	break
Else	finally	Elif	del	except
Global	for	If	from	import
Raise	try	Or	return	pass
Nonlocal	in	Not	is	lambda

Comments:

A computer program is a collection of instructions or statements.

A **Python** program is usually composed of multiple statements. Each statement is composed of one or a combination of the following:

1. [Comments](#)

2. Whitespace characters
3. Tokens

In a computer program, a comment is used to mark a section of code as non-executable.

Comments are mainly used for two purposes:

1. To mark a section of source code as non-executable, so that the Python interpreter ignores it.
2. To provide remarks or an explanation on the working of the given section of code in plain English, so that a fellow programmer can read the comments and understand the code.

In Python, there are two types of comments:

1. single-line comment : It starts with # (also known as the **hash** or **pound** character) and the content following # till the end of that line is a comment.
2. Docstring comment : Content enclosed between tripple quotes, either " or ". (We will learn about it later).

INPUT AND OUTPUT STATEMENTS

In Python, to read the input from the user, we have an in-built function called `input()`.

The syntax for `input()` function is :

```
input([prompt])
```

here, the **optional** prompt string will be printed to the user output and then will wait for the user input. The prompt string is to used to tell the user what to input.

Reading string inputs

Let us consider the below example:

```
name = input("Enter your Name: ")  
print("User Name:", name)
```

Sample input output of this program is as follows

Enter your Name: Anand Vihan

User Name: Anand Vihan

If the input is given as "**Jacob**", the result will be

User Name: Jacob

Eg:

Write a program to print your favorite place. At the time of execution, the program should print the message on the console as:

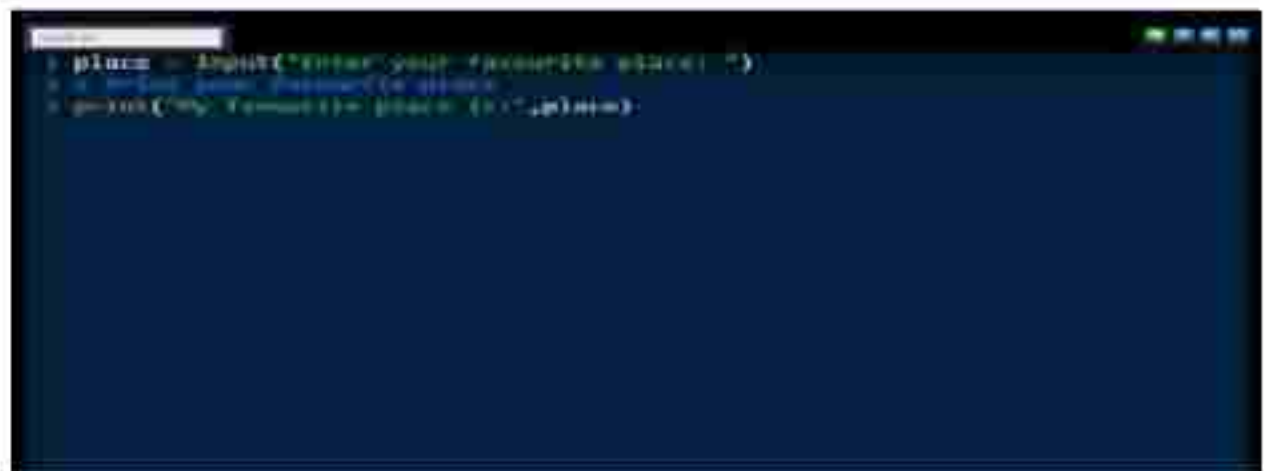
Enter your favourite place:

For example, if the user gives the input as:

Enter your favourite place: Hyderabad

then the program should print the result as:

My favourite place is: Hyderabad



```
> place = input("Enter your favourite place: ")
> if len(place) > 0:
>     print("My favourite is: " + place + ".")
```

UNDERSTANDING OUTPUT IN PYTHON:

We already discussed this function to print output on Screen.

It is **print()** function, which is a built-in function in Python. We can pass zero or more number of expressions separated by commas(,) to **print()** function.

The **print()** function converts those expressions into Strings and write

the result to standard output which then displays the result on Screen.

Let us consider an example of **print()** with multiple values with comma separator.

```
print("Hi", "Hello", "Python") # will print output as follows  
Hi Hello Python
```

Let us discuss another example:

```
a = 10  
b = 50  
print("A value is", a, "and", "B value is", b) # will print output as follows  
A value is 10 and B value is 50
```

Here we have assigned values 10 and 50 to a and b respectively.

The above **print()** statement consists of both strings and integer values.

Write a program to print your favourite programming language.

At the time of execution, the program should print the message on the console as:

```
Enter Language:
```

For example, if the user gives the input as:

```
Enter Language: Python
```

then the program should print the result as:

```
My Favourite Language is Python
```




Python Operators

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a particular programming language. Python provides a variety of operators described as follows.

- Arithmetic operators
- Comparison operators
- Assignment Operators
- Logical Operators
- Unary operators
- Bitwise Operators
- Membership Operators
- Identity Operators

1.Arithmetic Operators:

Arithmetic operators are used to perform arithmetic operations between two operands. It includes +(addition), -(subtraction), *(multiplication), /(divide), %(remainder), //(floor division), and exponent (**). In the table a=100,b=200.

Operator	Description	Example	Output
+	Addition: Adds the operands.	>>> print(a + b)	300
-	Subtraction: Subtracts operand on the right from the operand on the left of the operator.	>>> print(a - b)	-100
*	Multiplication: Multiplies the operands	>>> print(a * b)	70000
/	Division: Divides operand on the left side of the operator with the operand on its right. The division operator returns the quotient.	>>> print(b / a)	2.0
%	Modulus: Divides operand on the left side of the operator with the operand on its right. The modulus operator returns the remainder.	>>> print(b % a)	0
//	Floor Division: Divides the operands and returns the quotient. It also removes the digits after the decimal point. If one of the operands is negative, the result is floored (i.e. rounded away from zero towards negative infinity).	>>> print(12//5) >>> print(12.0//5.0) >>> print(-19//5) >>> print(-20.0//3)	2 2.0 -4 -7.0
**	Exponent: Performs exponential calculation, that is, raises operand on the right side to the operand on the left of the operator.	>>> print(a**b)	100 ²⁰⁰

I

2.Comparison Operators:

Comparison operators are used to comparing the value of the two operands and returns boolean true or false accordingly. The comparison operators are described in the following table a=5,b=6

Operator	Description	Example	Output
==	Returns True if the two values are exactly equal.	>>> print(a == b)	False
!=	Returns True if the two values are not equal.	>>> print(a != b)	True
>	Returns True if the value at the operand on the left side of the operator is greater than the value on its right side.	>>> print(a > b)	False
<	Returns True if the value at the operand on the right side of the operator is greater than the value on its left side.	>>> print(a < b)	True
>=	Returns True if the value at the operand on the left side of the operator is either greater than or equal to the value on its right side.	>>> print(a >= b)	False
<=	Returns True if the value at the operand on the right side of the operator is either greater than or equal to the value on its left side.	>>> print(a <= b)	True

3. Assignment operators

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

Operator	Description
<code>=</code>	It assigns the the value of the right expression to the left operand.
<code>+=</code>	It increases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if $a = 10$, $b = 20 \Rightarrow a+ = b$ will be equal to $a = a+ b$ and therefore, $a = 30$.
<code>--</code>	It decreases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if $a = 20$, $b = 10 \Rightarrow a- = b$ will be equal to $a = a- b$ and therefore, $a = 10$.
<code>*=</code>	It multiplies the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if $a = 10$, $b = 20 \Rightarrow a* = b$ will be equal to $a = a* b$ and therefore, $a = 200$.
<code>%=</code>	It divides the value of the left operand by the value of the right operand and assign the remainder back to left operand. For example, if $a = 20$, $b = 10 \Rightarrow a \% = b$ will be equal to $a = a \% b$ and therefore, $a = 0$.
<code>**=</code>	$a**=b$ will be equal to $a=a**b$, for example, if $a = 4$, $b = 2$, $a**=b$ will assign $4**2 = 16$ to a .
<code>//=</code>	$A//=b$ will be equal to $a = a// b$, for example, if $a = 4$, $b = 3$, $a//=b$

will assign $4//3 = 1$ to a.

4. Logical Operators:

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

Operator	Description
And	If both the expression are true, then the condition will be true. If a and b are the two expressions, $a \rightarrow \text{true}$, $b \rightarrow \text{true} \Rightarrow a \text{ and } b \rightarrow \text{true}$.
Or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, $a \rightarrow \text{true}$, $b \rightarrow \text{false} \Rightarrow a \text{ or } b \rightarrow \text{true}$.
Not	If an expression a is true then not (a) will be false and vice versa.

5. Unary Operators:

Unary operators act on single operands. Python supports unary minus operator. Unary minus operator is strikingly different from the arithmetic operator that operates on two operands and subtracts the second operand from the first operand. When an operand is preceded by a minus sign, the unary operator negates its value.

For example, if a number is positive, it becomes negative when preceded with a unary minus operator. Similarly, if the number is negative, it becomes positive after applying the unary minus operator. Consider the given example.

$b = 10$ $a = -(b)$

The result of this expression, is $a = -10$, because variable b has a positive value. After applying unary minus operator (-) on the operand b, the value becomes -10, which indicates it as a negative value.

6. Bitwise Operators:

The bitwise operators perform bit by bit operation on the values of the two operands.

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
(binary or)	The resulting bit will be 0 if both the bits are zero otherwise the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different otherwise the resulting bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

For example,

```
1.  if a = 7;
2.    b = 6;
3.  then, binary (a) = 0111
4.    binary (b) = 0011
5.    hence, a & b = 0011 (3 in decimal number)
6.    a | b = 0111 (7 in decimal number)
7.    a ^ b = 0100 (4 in decimal number)
8.    ~ a = 1000
9.    a << 2 = 11100 (28 in decimal number)
```

10. `a>>2=0001`(1 in decimal number)
Similarly 5 binary equivalent is 0101 and 3 is 0011.

7. Membership Operators:

Python supports two types of membership operators-in and not in. These operators, test for membership in a sequence such as strings, lists, or tuples.

in Operator: The operator returns true if a variable is found in the specified sequence and false otherwise.

For example, `a=6`

`nums=[1,4,3,7,6]`

`a in nums` returns True, if `a` is a member of `nums`.

not in Operator: The operator returns true if a variable is not found in the specified sequence and false otherwise.

For example, `a=99`

`Nums=[1,3,5,7,2,0]`

`a not in nums` returns True, if `a` is not a member of `nums` otherwise returns False.

8. Identity Operators

is Operator: Returns true if operands or values on both sides of the operator point to the same object (*Memory location*)and False otherwise. For example, if `a is b` returns True , if `id(a)` is same as `id(b)` otherwise returns False. `id(a)` gives memory location of `a`

is not Operator: Returns true if operands or values on both sides of the operator does not point to the same object(*Memory location*) and False otherwise. For example, if `a is not b` returns 1, if `id(a)` is not same as `id(b)`.

Operator Precedence

The precedence of the operators is important to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in python is given below.

Operator	Description
<code>**</code>	The exponent operator is given priority over all the others used in the expression.

<code>- + -</code>	The negation, unary plus and minus.
<code>* / % //</code>	The multiplication, divide, modules, reminder, and floor division.
<code>+ -</code>	Binary plus and minus
<code>>> <<</code>	Left shift and right shift
<code>&</code>	Binary and.
<code>^ </code>	Binary xor and or
<code>< <= > >=</code>	Comparison operators (less then, less then equal to, greater then, greater then equal to).
<code><= == !=</code>	Equality operators.
<code>= %= /= //=-= += *= **=</code>	Assignment operators
<code>is is not</code>	Identity operators
<code>in not in</code>	Membership operators
<code>not or and</code>	Logical operators

Decision Control Statements

A *control statement* is a statement that determines the control flow of a set of instructions, i.e., it decides the sequence in which the instructions in a program are to be executed.

Types of Control Statements —

- Sequential Control: A Python program is executed sequentially from the first line of the program to its last line.
- Selection Control: To execute only a selected set of statements.
- Iterative Control: To execute a set of statements repeatedly.

Selection Control Statements:

The selection statements in Python are

1. simple if or if statement
2. if else statement
3. nested if else statement
4. if elif else statement

1. simple if statement or if statement:

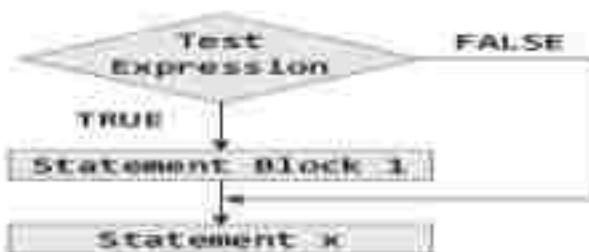
Syntax:

```
if(expression):  
    Statementblock  
statementx
```

Explanation:

- Expression is also called as condition.
- Statement block may contain single statement or more statements.
- The expression in the if statement should be either relational (eg: $a > 0$ or $a > b$) or logical (eg: $a > b \&\& a > c$) expression.
- The expression is evaluated first and if it results true then the statementblock under if condition executed and come out of if statement and executes the remaining statements of the program.
- If the expression results false then it skips the if statement part and executes the remaining statements of the program.

Flow chart:



Example 1:

```
x = 10      #initialize the value of x
if(x>0):    #test the value of x
    x = x+1  #Increment the value of x if it is > 0
print(x)    #Print the value of x
```

OUTPUT

x = 11

Example 2:

Write a program to find small number of two numbers

```
a=int(input("enter two numbers"))
b=int(input("enter two numbers"))
if(a<b):
    print(a,"is small number")
if(b<a):
    print(b,"is small number")
```

Note: This program is an example of *multiple if statements*

Exercise:

1. Write a Program to find biggest number of three numbers using if statement.
2. Write a Program to find whether the given character is vowel or digit using if statement.

2. if else statement:

It's another kind of if statement. It is a two way selection statement.

Syntax:

```
if(expression):
    statementblock1

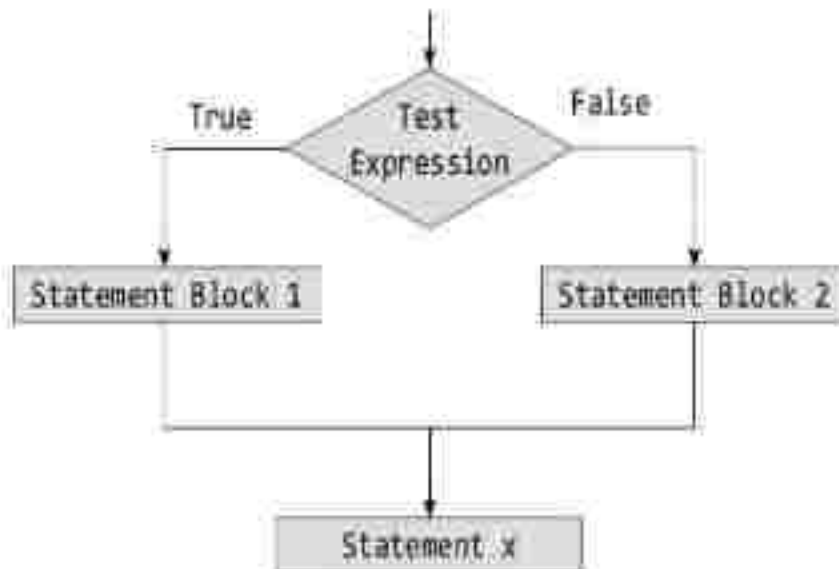
else:
    statementblock2
statementx #this statement is optional
```

Explanation:

- In this statement first expression will be evaluated. If the expression evaluates to true then statementblock1 under if condition executed and executes the remaining statements of the program.

- Otherwise statementblock2 under else part will be executed and executes the remaining statements of the program.

Flow chart:



Programs:

1. Write a Program to Input age and check whether the person is eligible to vote or not

```

age = int(input("Enter the age : "))
if(age>18):
    print("You are eligible to vote")
else:
    yrs = 18 - age
    print("You have to wait for another " + str(yrs) + " years to cast your vote")
  
```

OUTPUT

```

Enter the age : 18
You have to wait for another 8 years to cast your vote.
  
```

2. Write a program to find small number of two numbers using if else statement

```

a=int(input("enter two numbers"))
b=int(input("enter two numbers"))
if(a<b):
    print(a,"is small number")
if(b<a):
  
```

```
print(b,"is small number")
```

3. Program to check whether the given character is vowel or not using if else statement.

```
ch=input("enter a character" )  
if(ch=='a' or ch=='A' or ch=='e' or ch=='E' or ch=='i' or ch=='I' or ch=='o' or  
ch=='O' or ch=='u' or ch=='U'):  
    print(ch,"is a vowel character")  
else:  
    print(ch,"is not a vowel character")
```

Exercise:

1. Write a Program to check whether the given number is even number or not
2. Write a Program to input employee salary .If the employee is male(m) then given 10% bonus otherwise give 25% bonus on salary.Display the final salary will get by that employee.

3.Nested if else statement or nested if statement:

An if else statement or if statement contains another if else statement within it then such if else called nested if else statement or nested if statement. The nested if else can contain any number of if statements or if else statements either under if part or else part.This statement used when we want to specify more number conditions to provide more options. It is also called as multi way selection statement.

Syntax:

```
if( expression1):
```

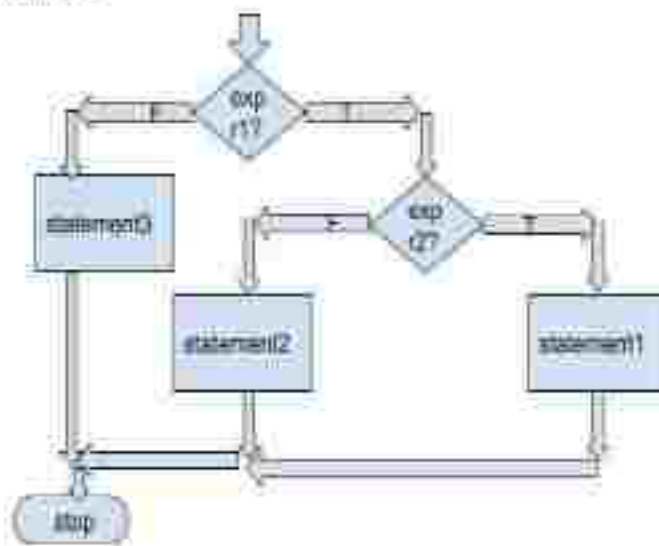
```
    if(expression2):  
        Statementblock1
```

```
    else:  
        Statementblock2
```

```
else:  
    if(expression3):  
        Statementblock3
```

```
    else:  
        Statementblock4
```

Flow Chart:



In the above flowchart, statement 1, statement 2 are representing the corresponding blocks.

Example Programs:

1. Write a program to find biggest number among three numbers.

```
a=int(input("enter first number"))
b=int(input("enter second number"))
c=int(input("enter third number"))
```

```
if(a>b):
    if(a>c):
        print(a," is small number")
    else:
        print(c,"is small number")
```

```
else:
    if(b>c):
        print(b," is small number")
    else:
        print(c,"is small number")
```

2. Program to check whether the given number is zero ,positive or negative using nested if statement concept.

```
n=int(input("enter a number"))
if(n>=0):
    if(n==0):
        print(n,"is zero")
    else:
        print(n,"is a positive number")
else:
    print(n,"is a negative number")
```

4. if elif else statement:

It is also called multi way selection statement. It is an alternative to nested if statement.

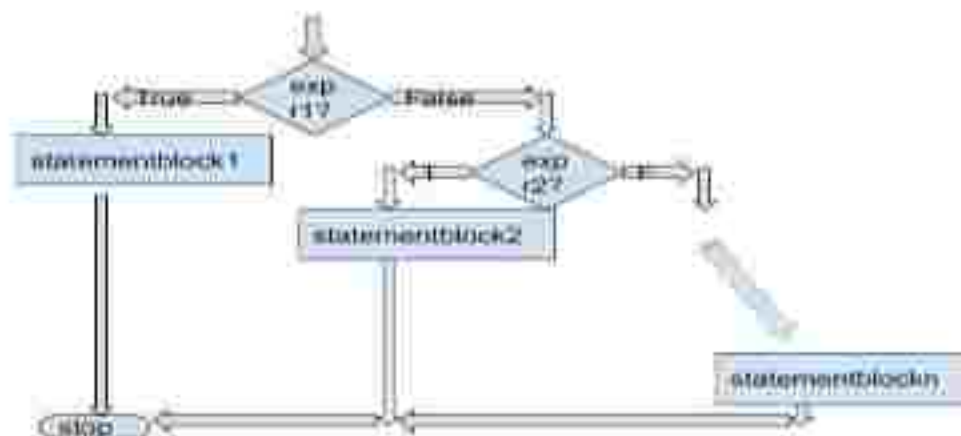
Syntax:

```
if(expression1):
    Statementblock1;
else if (expression2):
    Statementblock2;
    :
    :
else:
    Statementblockn;
```

Explanation:

In this the expressions are evaluated from top to bottom. Whenever the expression evaluates true then the corresponding statement block will be executed and come out of the if elif else statement process. If that expression is false we move to next expression and evaluate it. if all the expressions evaluate to false then last else part will be executed i.e. statementblockn . The statement may have single or multiple statements.

Flow chart:



Example Programs:

1. Write a Program to check whether the given number is zero, positive or negative using if elif else statement.

```

num = int(input("Enter any number : "))
if(num==0):
    print("The value is equal to zero")
elif(num>0):
    print("The number is positive")
else:
    print("The number is negative")
  
```

OUTPUT

```

Enter any number : -10
The number is negative
  
```

2. Write a Program to accept five subject marks and then calculate the total, average. Display the total marks, average marks and result based on the following criteria.

<u>Average</u>	<u>Result</u>
>=70	distinction
60-70	first class
50-60	second class

40-50

third class

Answer:

```
a=int(input("enter first subject marks"))
b=int(input("enter second subject marks"))
c=int(input("enter third subject marks"))
d=int(input("enter fourth subject marks"))
e=int(input("enter fifth subject marks"))
total=a+b+c+d+e
avg=total/5

if(a<35 or b<35 or c<35 or d<35 or e<35):
    print(" fail")
elif(avg>=70):
    print("distinction")
elif(avg>=60):
    print(" first class")
elif(avg>=50):
    print(" second class")
else :
    print(" third class")
```

3. Write a program to find the roots of a quadratic equation.

```
a=int(input("enter a number"))
b=int(input("enter b number"))
c=int(input("enter c number"))
d=b*b-4*a*c;
if(d==0):
    print(" roots are real and equal")
    r1=-b/(2*a);
    r2=r1;
    print(" root1 value=",r1,"root2 value=",r2)

elif(d>0):

    print(" roots are real and unequal")
    r1=(-b+d**0.5)/(2*a);
    r2=(-b-d**0.5)/(2*a);
```

```

        print("root1 value=",r1,"root2 value=",r2)
    else:

        print("roots are imaginary")

```

4. Write a program to find biggest number among three numbers.

```

a=int(input("enter first number"))
b=int(input("enter second number"))
c=int(input("enter third number"))

if(a>b and a>c):
    print(a,"is biggest number")

elif(b>c):
    print(b," is biggest number")
else:
    print(c,"is biggest number")

```

5. Read the user input to enter a temperature in Celsius. The program should print a message based on the temperature:

- If the temperature is less than -273.15, print that ***"The temperature is invalid"*** because it is below absolute zero.
- If it is exactly -273.15, print that ***"The temperature is absolute 0"***.
- If the temperature is between -273.15 and 0, print that ***"The temperature is below freezing"***.
- If it is 0, print that ***"The temperature is at the freezing point"***.
- If it is between 0 and 100, print that ***"The temperature is in the normal range"***.
- If it is 100, print that ***"The temperature is at the boiling point"***.
- If it is above 100, print that ***"The temperature is above the boiling point"***.

Answer:

```

t=float(input("enter a temperature in Celsius"))
if(t<-273.15):
    print("The temperature is invalid")
elif(t== -273.15):
    print("The temperature is absolute 0")
elif(t>-273.15 and t<0):
    print("The temperature is below freezing")
elif(t==0):
    print("The temperature is at freezing point")
elif(t>0 and t<100):
    print("The temperature is in the normal range")

```

```
elif(t==100):
    print("The temperature is at boiling point")
else:
    print("The temperature is at boiling point")
```

Exercise:

1. Write a program to display the day of a week based on number.
2. Write a program to generate electricity bill amount based on the following constraints

Consumed Units	amount
<100	Rs. 1.60/unit
100-200	Rs.2.35/unit
201-400	Rs.3.40/unit
>400	Rs.5.25/unit

3. Write a Program to input a digit and then display that digit in word using if elif else statement.

Important Topics:

1. Explain various conditional or selection statements(if,if else,nested if ,if elif else statement) with an example. (*Hint:For every statement,write syntax,explanation,draw a flow chart(diagram) and small example program*)
2. Programs on if else and if elif else statements.

Loop statements or iterative statements or Repetitive statements:

Looping means executing a statement or set of statements repeatedly until a specified condition becomes false.

Generally the loop control statements has two things ,one thing is body of the loop which to be executed repeatedly and second thing is the condition which is tested for termination of the loop.

Python supports two types of loop statements

- 1) While loop
- 2) For loop

While loop:

loop control statement contains three things

- Loop variable initialization:loop variable has to be initialized with initial value or final value.

```
Eg: i=5;
    j=i+1
    i=n
```


- Loop condition/condition: it can be a relational expression or logical expression
Eg: $i > 0$
 $i > 10 \ \& \ j < 100$
- Loop variable increment/decrement operation(Update Statement): In this we can increment or decrement the value of variable using increment/decrement operators.
Eg: $i = i + 1$
 $j = j - 1$
 $n /= -10$ etc

Syntax:

initialization

while (Expression):

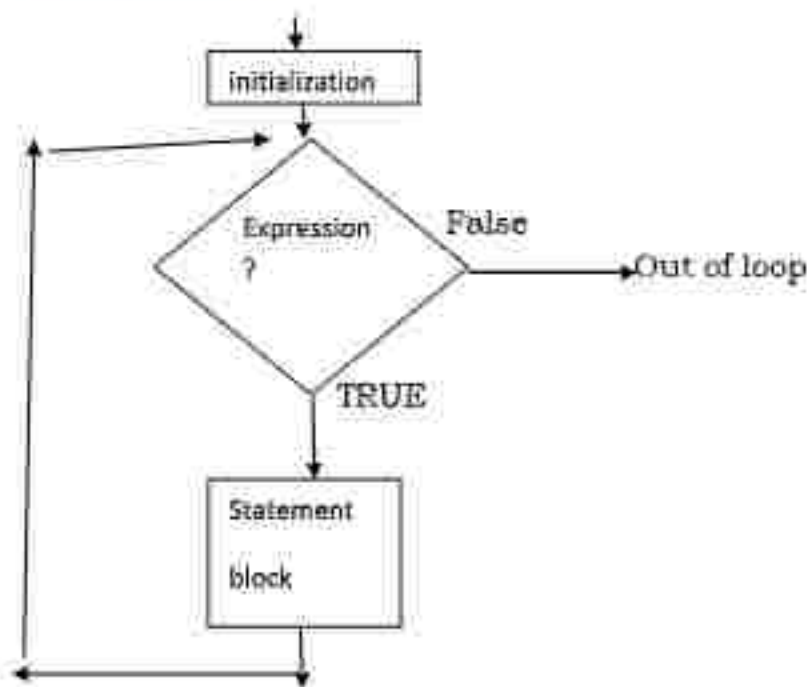
statementblock

#this is called body of loop

Explanation:

In the while loop, first the initialization statement gets executed, then condition or expression is verified if that is true then all the statements in statement block (body of while loop) gets executed. After that again the condition will be verified if it's false then we stop the process otherwise the execution continues.

Flow Chart:



Example Programs:

1. Write a Program to display the 1 to 5 numbers line by line using while loop

```
i=1 #This is called initialization
while(i<=5):
    print(i)
    i=i+1 # This is called Updation
```

Output:

```
1
2
3
4
5
```

2. Write a Program to print 0 to 10 in the same line or single line using while loop

```
i = 0
while(i<=10):
    print(i,end=" ")
    i = i+1
```

OUTPUT

```
0 1 2 3 4 5 6 7 8 9 10
```

3. Write a program to find sum of n natural numbers and average value

```
n=int(input("enter n value"))
i=1
sum=0
while(i<=n):
    sum=sum+i
    i=i+1

print("sum of ",n," natural numbers value=",sum)
avg=sum/n
print("average value=",avg)
```

4. Write a program to find the sum of the digits of a given number.

```
n=int(input("enter n value"))
t=n #assign n value to t
sum=0
while(n!=0):
    d=n%10
    sum=sum+d
    n=n//10

print("sum of the digits of number",t,"is",sum)
```

Input:

Enter n value 156

Output:

Sum of the digits of number 156 is 12

5. Write a program to check whether the given number is Armstrong number or not(Hint: if number is equal to sum of the digits raised to the power of number of digits in that number.Ex:1) $153=1^3+5^3+3^3$

$$2)1634=1^4+6^4+3^4+4^4$$

n=input("enter n value") # n holds a string even you enter 67 it will be stored as "67"

l=len(n) # len function gives length(number of characters) of the string

sum=0

n=int(n) # converting string into integer

t=n

```
while(n!=0):
    d=n%10
    sum=sum+d**l
    n=n//10
```

print("sum of the digits of number",t,"is",sum)

if(sum==t):

print(t,"is an Armstrong number")

else:

print(t,"is not an Armstrong number")

Input:

Enter n value 156

Output:

Sum of the digits of number 156 is 12

156 is not an Armstrong number

6. Write a program to find the reverse number of a given number

```
n=int(input("enter n value"))
r=n #assign n value to t
sum=0
while(n!=0):
    d=n%10
    sum=sum*10+d
    n=n//10

print(" The reverse number of given number",t,"is",sum)
```

Input:

Enter n value 156

Output:

The reverse number of given number 156 is 651

Exercise:

1. Write a program to check whether the given number is palindrome number or not.
2. Write a program to check whether the given number is perfect number or not.
(Hint: If a number equals to sum of it's factors (except the number itself) then that number is called perfect number. Ex: 6 is a input number and it's factors are 1,2,3,6 --6 is a perfect number because $6=1+2+3$)
3. Write a program to the given decimal number into binary number using while loop. (Ex: if decimal number is 5 then it's equivalent binary number is 101, Hint: use %, // and denominator is 2)
4. Write a program to the given binary number into decimal number using while loop. (Ex: if binary number is 101 then it's equivalent decimal number is 5)
5. Write a Program to display all the numbers divisible by 3, divisible by 5, divisible by both 3 and 5 from 1 to n numbers

Solutions for your reference

Solution for 3:

#Program to convert decimal number to binary number

```
n=int(input("enter a decimal number"))
```

```
t=n
```

```
sum=0
```

```
i=0
```

```
while(n!=0):
```

```
    r=n%2
```

```
    sum=sum + r*10**i
```

```
    n=n//2
```

```
    i=i+1
```

```
print(t,"binary equivalent is",sum)
```

Input:

enter a decimal number34

Output:

34 binary equivalent is 100010

Solution for 4:

#Program to convert binary number to decimal number

```
n=int(input("enter a binary number"))
```

```
t=n
```

```
sum=0
```

```
i=0
```

```
while(n!=0):
```

```
    r=n%10
```

```
    sum=sum+ r*2**i
```

```
    n=n//10
```

```
    i=i+1
```

```
print(t,"decimal equivalent is",sum)
```

input:

enter a binary number1011

Output:

1011 decimal equivalent is 11

Note:

range() function: This function used to generate a sequence of integer numbers in a range. It's an inbuilt function like input(), print() functions.

Syntax:

```
range(start, stop, step)
```

start: it's optional and it's an integer value. It indicates starting point of the sequence. By default its value is 0 (zero)

stop: it's required and it's an integer value. It's an ending of the sequence but this value will not be included in the sequence.

Step: It's an integer value that determines the increment between each integer in sequence. Default increment value is 1.

Ex:

1. range(5) —it generates sequence of numbers from 0 to 4
2. range(0,5)—it generates sequence of numbers from 0 to 4
3. range(n)— it generates sequence of numbers from 0 to n-1
4. range(5,10)— it generates sequence of numbers from 5 to 9
5. range(5,10,2)— it generates sequence of numbers 5, 7, 9
6. range(-2,2,2)— It generates sequence of numbers -2,0
7. range(-100,-95,2)— it generates sequence of numbers -100,-98,-96

FOR LOOP:

- Used to execute group of statements depending upon number of elements in the sequence.
- It can work with sequence like range, string, list, tuple etc

- **Syntax:**

```
for var in sequence:  
    statementblock
```

Explanation:

- The first element of the sequence is assigned to variable var and then statement block executed.

- Next the second element of the sequence is assigned to variable var and then statement block executed.
- This process will be continued until the entire sequence is exhausted or completed Means for each element of the sequence ,the statement block executed once.

Examples:

1. for i in range(1,4):

```
    print(i)
    print("End")
```

output: 1

```
    End
    2
    End
    3
    End
```

2. for i in range(1,4):

```
    print(i)
print("End")
```

output: 1

```
    2
    3
    End
```

3. for i in range(-100,-95,2):

```
    print(i)
```

output: -100
-98
-96

Programs:

1. Write a program to find factorial of a number

```
#Program to find the factorial of a number
n=int(input("enter a number"))
f=1
for i in range(1,n+1):
    f=f*i
print("factorial of ",n,"is",f)
```

input:

enter a number 6

Output:

factorial of 6 is 720

2. Write a program to check whether the given number is prime number or not

```
n=int(input("enter a number"))
c=0
for i in range(1,n+1):
    if(n%i==0):
        c=c+1

if(c==2):
    print(n,"is a prime number")
else:
    print(n,"is not a prime number")
```

input:

enter a number 6

Output:

6 is not a prime number

3. Write a program to print multiplication table of a given number

```
n=int(input("enter a number"))
print("multiplication table of",n)
print("*****")
```

```

for i in range(1,11):
    print(n,"X",i,"=",n*i)
print("*****")

```

input:

enter a number 5

Output:

multiplication table of 5

5 X 1 = 5

5 X 2 = 10

5 X 3 = 15

5 X 4 = 20

5 X 5 = 25

5 X 6 = 30

5 X 7 = 35

5 X 8 = 40

5 X 9 = 45

5 X 10 = 50

4. Write a Program to find the sum of the following series

$\frac{1}{2} + \frac{2}{3} + \frac{3}{4} + \dots + \frac{n}{n+1}$

#Program to find the sum of the following series

$\frac{1}{2} + \frac{2}{3} + \frac{3}{4} + \dots + \frac{n}{n+1}$

```
n=int(input("enter n value"))
```

```
s=0
```

```
for i in range(1,n+1):
```

```
    s=s+1/(i+1)
```

```
print("sum of the series is",s)
```

input:

enter n value 4

Output:

sum of the series is 2.716666666666667
--

Exercise:

1. Write a Program to check whether the given number is perfect number or not
2. Write a Program to print all the numbers divisible by 7 and 9 in a range 1 to n (n value supposed to be entered by the user)
3. Write a Program to print all the even numbers in a range 1 to n (n value supposed to be entered by the user)
4. Write a program to find the sum of the following series

$$1 + 1/2 + 1/3 + 1/4 + \dots + 1/n$$

Nested Loops:

If any loop contains any other loop of same or different then it's called nested loop. Any loop can contain any number of other loops.

Python allows its users to have nested loops, that is, loops that can be placed inside other loops. Although this feature will work with any loop like while loop as well as for loop.

A for loop can be used to control the number of times a particular set of statements will be executed. Another outer loop could be used to control the number of times that a whole loop is repeated.

Loops should be properly indented to identify which statements are contained within each for statement.

Examples:

1)

```
while(condition1):  
    --- #some statement(s)  
    while(condition2):  
        --- #some statement(s)
```

2)

```
while(condition1):  
    --- #some statement(s)  
    for var in sequence:  
        --- #statement(s)
```

-----#some statement(s)

-----#statement

3)

```
for var in sequence:  
    -----# statement(s)  
    for var in sequence:  
        -----# statement(s)
```

4)

```
for var in sequence:  
    -----# statement(s)  
    while(condition):  
        -----#statement
```

Example programs;

1) Write a Program to print all the prime numbers from 1 to 1000

```
for i in range(1,1001):  
    n=i  
    c=0  
    for i in range(1,n+1):  
        if(n%i==0):  
            c=c+1  
  
    if(c==2):  
        print(n,"is a prime number")
```

2) Write a Program to check whether the given number is strong number or not

program to check whether the given number is strong number or not

```
num=int(input("enter a number"))
```

```
t=num
```

```
s=0
```

```
while(num!=0):
```

```

d=num%10
n=d
f=1
for i in range(1,n+1):
    f=f*i
s=s+f
num=num//10

if(t==s):
    print(t,"is a strong number")
else:
    print(t,"is not a strong number")

```

input:

enter a number 145

Output:

145 is a strong number

3) Write a Program to print the following pattern

```

*****
*****
*****
*****
*****

```

```

for i in range(1,6): #i indicates row
    for j in range(1,6):
        print("*",end=" ")
    print()

```

4) Write a Program to print the following pattern

```
12345
12345
12345
12345
12345
```

```
for i in range(1,6): #i indicates row
    for j in range(1,6): # j indicates cloumn
        print(j,end=" ")
    print()
```

5) Write a Program to print the following pattern

```
11111
22222
33333
44444
55555
```

```
for i in range(1,6): #i indicates row
    for j in range(1,6):
        print(i,end=" ")
    print()
```

6) Write a Program to print the following pattern

```
*
**
***
****
*****
```

```
for i in range(1,6): #i indicates row
    for j in range(1,i+1):
        print("*",end=" ")
    print()
```

Exercise:

1) Write a Program to print the following pattern

```
1
12
123
```

1234

12345

- 2) Write a Program to print the following pattern

1

22

333

4444

55555

- 3) Write a Program to print the following pattern

12345

1234

123

12

1

- 4) Write a Program to print the following pattern

12345

2345

345

45

5

- 5) Write a Program to print the following pattern

54321

5432

543

54

5

- 6) Write a Program to print the following pattern

54321

4321

321

21

1

- 7) Write a Program to print the following pattern

```

      *
    * *
  * * *
* * * *
```

```
#pyramid
```

```
***
 *
 * *
* * *
* * * *
```

```
***
```

```
for i in range(1,5):
    for s in range(1,5-i):
        print(" ",end="")
    for j in range(1,i+1):
        print("*",end=" ")
    print()
```

Output:

```

 *
 * *
* * *
* * * *
```

8) Write a Program to print the following pattern

```
*****
*               *
*               *
*               *
*               *
*               *
*****
```

#to print square pattern


```

for i in range(1,11):
    for j in range(1,11):
        if(i==1 or j==1 or i==10 or j==10):
            print("**",end=" ")
        else:
            print(" ",end=" ")
    print()

```

9) Write a Program to print the following pattern

```

* * * * *
* *       *
*  *       *
*   *       *
*    *       *
*     * *
* * * * *

```

10) Write a Program to print the following pattern

```

* * * * *
*
*
*
* * * * *

```

#To print C pattern

```

for i in range(1,9):
    for j in range(1,6):
        if(i==1 or i==8 or j==1):
            print("**",end=" ")
        else:
            print(" ",end=" ")
    print()

```

11) Write a Program to print the following pattern

```

*****
*
*
*
*****
  *
  *
  *
*****

```

12) Write a Program to print the following pattern

```

      1
     1 2
    1 2 3
   1 2 3 4

```

```

#To print the pattern
for i in range(1,5):
    for s in range(1,5-i):
        print(" ",end="")
    for j in range(1,i+1):
        print(j,end="")
    print()

```

13. Write a Program to print the following pattern

```

      1
     1 2 1
    1 2 3 2 1
   1 2 3 4 3 2 1

```

14. Write a Program to print the Heart symbol following pattern

15. Write a Program to print the following pattern

```

123454321
1234 4321
123  321
12   21
1    1
12   21
123  321
1234 4321
123454321

```

16. Write a Program to print the following pattern

```

      *
    * *
  * * *
* * * *
  * * *
    * *
      *
  
```

Break, continue and Pass statements:

Break Statement:

The *break* statement is used to terminate the execution of the nearest enclosing loop in which it appears. The break statement is widely used with for loop and while loop. When interpreter encounters a break statement, the control passes to the statement that follows the loop in which the break statement appears.



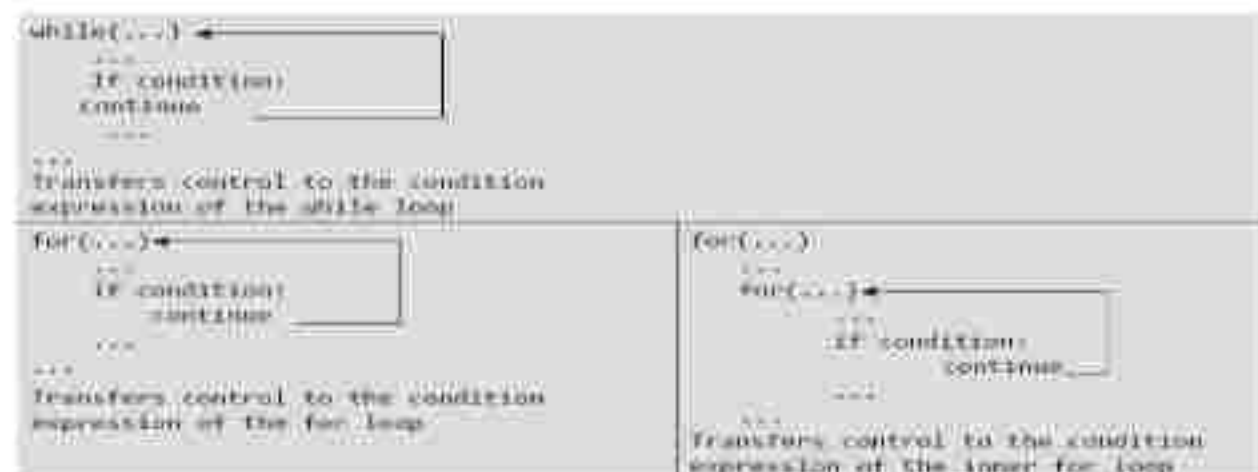
Example:

```

1 2 3
while i <= 10:
    print(i, end=" ")
    if i==5:
        break
    i = i+1
print("\n Done")
OUTPUT
1 2 3 4 5
Done
  
```

The Continue Statement

Like the break statement, the continue statement can only appear in the body of a loop. When the interpreter encounters a continue statement then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the loop-continuation portion of the nearest enclosing loop.



Example:

```
for i in range(1,11):
    if(i==5):
        continue
    print(i, end=" ")
print("\n Done")
```

OUTPUT

```
1 2 3 4 6 7 8 9 10
Done
```

Pass Statement:

Pass statement is used when a statement is required syntactically but no command or code has to be executed. It specifies a *null* operation or simply No Operation (NOP) statement. Nothing happens when the pass statement is executed.

Difference between comment and pass statements in Python programming, pass is a null statement. The difference between a comment and pass statement is that while the interpreter ignores a comment entirely, pass is not ignored. Comment is not executed but pass statement is executed but nothing happens.

Example:

```
for letter in "HELLO":  
    pass      #The statement is doing nothing  
    print("Pass : ", letter)  
print("Done")
```

OUTPUT

```
Pass : H  
Pass : E  
Pass : L  
Pass : L  
Pass : O  
Done
```

The Else Statement Used With Loops

Unlike C and C++, in Python you can have the *else* statement associated with a loop statements. If the *else* statement is used with a *for* loop, the *else* statement is executed when the loop has completed iterating. But when used with the *while* loop, the *else* statement is executed when the condition becomes false.

Examples:

```
for letter in "HELLO":  
    print(letter, end=" ")  
else:  
    print("Done")
```

OUTPUT

```
H E L L O Done
```

```
i = 1  
while(i < 0):  
    print(i)  
    i = i - 1  
else:  
    print(i, "is not negative so  
loop did not execute")
```

OUTPUT

```
1 is not negative so loop did not execute
```


STRINGS

INTRODUCTION:

Python treats strings as contiguous series of characters delimited by single, double or even triple quotes. Python has a built-in string class named "str" that has many useful features. We can simultaneously declare and define a string by creating a variable of string type.

This can be done in several ways which are as follows:

```
name = "India"
```

```
graduate = 'N'
```

```
country = name
```

```
Branch=""" computer
```

```
science
```

```
and engineering""
```

NOTE: we can use triple single quotes(''') also for writing the strings.

Indexing:

Individual characters in a string are accessed using the subscript ([]) operator. The expression in brackets is called an index. The index specifies the character we want to access from the given set of characters in the string.

The index of the first character is 0 and that of the last character is n-1 where n is the number of characters in the string. If you try to exceed the bounds (below 0 or above n-1), then an error is raised.

Traversing a String:

A string can be traversed by accessing character(s) from one index to another. For example, the following program uses indexing to traverse a string from first character to the last.

Example1:

```
message = "Hello!"
index = 0
for i in message:
    print("message[", index, "] = ", i)
    index += 1
```

OUTPUT

```
message[ 0 ] = H
message[ 1 ] = e
message[ 2 ] = l
message[ 3 ] = l
message[ 4 ] = o
message[ 5 ] = !
```

Example2:

Message="WELCOME"

for i in Message:

```
    print(i,end="")
```

Output:

WELCOME

Concatenating, Appending and Multiplying Strings

EXAMPLE for CONCATNATION:

```
str1 = "Hello "
str2 = "World"
str3 = str1 + str2
print("The concatenated string is : ", str3)
```

OUTPUT

The concatenated string is : Hello World

EXAMPLE for APPENDING:

```
str = "Hello, "  
name = input("\n Enter your name : ")  
str += name  
str += ". Welcome to Python Programming."  
print(str)
```

OUTPUT

```
Enter your name : Arnav  
Hello, Arnav. Welcome to Python Programming.
```

EXAMPLE for MULTIPLYING:

```
str = "Hello"  
print(str * 3)
```

OUTPUT

```
Hello Hello Hello
```

Strings are Immutable:

Python strings are immutable which means that once created they cannot be changed. Whenever you try to modify an existing string variable, a new string is created.

Example:

```
str1 = "Hello"  
print("Str1 is : ", str1)  
print("ID of str1 is : ", id(str1))  
str2 = "World"  
print("Str2 is : ", str2)
```

```

print("ID of str1 is : ", id(str2))
str1 += str2
print("Str1 after concatenation is : ", str1)
print("ID of str1 is : ", id(str1))
str3 = str1
print("str3 = ", str3)
print("ID of str3 is : ", id(str3))

```

OUTPUT

```

Str1 is : Hello
ID of str1 is : 45095344
Str2 is : World
ID of str1 is : 45095312
Str1 after concatenation is : Helloworld
ID of str1 is : 43861792
str3 = Helloworld
ID of str3 is : 43861792

```

Built-in String Methods and Functions

Function	Usage	Example
capitalize()	This function is used to capitalize first letter of the string.	<pre>str = "hello" print(str.capitalize())</pre> <p>OUTPUT</p> <p>Hello</p>
center(width, fillchar)	Returns a string with the original string centered to a total of width columns and filled with fillchar in columns that do not have characters.	<pre>str = "hello" print(str.center(10, '*'))</pre> <p>OUTPUT</p> <p>*xhellox**</p>

<code>count(str, beg, end)</code>	Counts number of times str occurs in a string. You can specify beg as 0 and end as the length of the message to search the entire string or use any other value to just search a part of the string.	<pre>str = "he" message = "helloworldhelloworld" print(message.count(str,0, len(message)))</pre> <p>OUTPUT</p> <p>3</p>
<code>endswith(suffix, beg, end)</code>	Checks if string ends with suffix; returns True if so and False otherwise. You can either set beg = 0 and end equal to the length of the message to search entire string or use any other value to search a part of it.	<pre>message = "She is my best friend" print(message.endswith("end", 0, len(message)))</pre> <p>OUTPUT</p> <p>True</p>
<code>find(str, beg, end)</code>	Checks if str is present in string. If found it returns the position at which str occurs in string, otherwise returns -1. You can either set beg = 0 and end equal to the length of the message to search entire string or use any other value to search a part of it.	<pre>message = "She is my best friend" print(message.find("my",0, len(message)))</pre> <p>OUTPUT</p> <p>7</p>
<code>index(str, beg, end)</code>	Same as find but raises an exception if str is not found.	<pre>message = "She is my best friend" print(message.index("mine", 0, len(message)))</pre> <p>OUTPUT</p> <p>ValueError: substring not found</p>
<code>rfind(str, beg, end)</code>	Same as find but starts searching from the end.	<pre>str = "Is this your bag?" print(str.rfind("is", 0, len(str)))</pre> <p>OUTPUT</p> <p>5</p>
<code>rindex(str, beg, end)</code>	Same as rindex but start searching from the end and raises an exception if str is not found.	<pre>str = "Is this your bag?" print(str.rindex("you", 0, len(str)))</pre> <p>OUTPUT</p> <p>8</p>

<code>isalnum()</code>	Returns True if string has at least 1 character and every character is either a number or an alphabet and False otherwise.	<pre>message = "JamesBond007" print(message.isalnum())</pre> <p>OUTPUT</p> <p>True</p>
<code>isalpha()</code>	Returns True if string has at least 1 character and every character is an alphabet and False otherwise.	<pre>message = "JamesBond007" print(message.isalpha())</pre> <p>OUTPUT</p> <p>False</p>
<code>isdigit()</code>	Returns True if string contains only digits and False otherwise.	<pre>message = "007" print(message.isdigit())</pre> <p>OUTPUT</p> <p>True</p>
<code>islower()</code>	Returns True if string has at least 1 character and every character is a lowercase alphabet and False otherwise.	<pre>message = "Hello" print(message.islower())</pre> <p>OUTPUT</p> <p>False</p>
<code>isspace()</code>	Returns True if string contains only whitespace characters and False otherwise.	<pre>message = " "</pre> <pre>print(message.isspace())</pre> <p>OUTPUT</p> <p>True</p>
<code>isupper()</code>	Returns True if string has at least 1 character and every character is an upper case alphabet and False otherwise.	<pre>message = "HELLO" print(message.isupper())</pre> <p>OUTPUT</p> <p>True</p>
<code>len(string)</code>	Returns the length of the string.	<pre>str = "Hello" print(len(str))</pre> <p>OUTPUT</p> <p>5</p>
<code>ljust(width[, fillchar])</code>	Returns a string left-justified to a total of width columns. Columns without characters are padded with the character specified in the fillchar argument.	<pre>str = "Hello" print(str.ljust(10, '*'))</pre> <p>OUTPUT</p> <p>Hello*****</p>
<code>rjust(width[, fillchar])</code>	Returns a string right-justified to a total of width columns. Columns without characters are padded with the character specified in the fillchar argument.	<pre>str = "Hello" print(str.rjust(10, '*'))</pre> <p>OUTPUT</p> <p>*****Hello</p>

Slice Operation

A substring of a string is called a *slice*. The slice operation is used to refer to sub-parts of sequences and strings. You can take subset of string from original string by using `| |` operator also known as *slicing operator*.

Index from the start	P	Y	T	H	O	N	Index from the end
	0	1	2	3	4	5	
	-6	-5	-4	-3	-2	-1	

Examples:

```
str = "PYTHON"
print("str[1:5] = ", str[1:5])  # characters starting at index 1 and extending up
                                # to but not including index 5
print("str[:6] = ", str[:6])   # defaults to the start of the string
print("str[1:] = ", str[1:])   # defaults to the end of the string
print("str[:] = ", str[:])     # defaults to the entire string
print("str[1:20] = ", str[1:20]) # an index that is too big is truncated down to
                                # length of the string
```

OUTPUT

```
str[1:5] = YTHO
str[:6] = PYTHON
str[1:] = YTHON
str[:] = PYTHON
str[1:20] = YTHON
```

Programming Tip: Python does not have any separate data type for characters. They are represented as a single character string.

Specifying Stride or step while Slicing Strings :

In the slice operation, you can specify a third argument as the *stride*, which refers to the number of characters to move forward after the first character is retrieved from the string. By default the value of stride is 1, so in all the above examples where he had not specified the stride, it used the value of 1 which means that every character between two index numbers is retrieved.

Example:

```
str = "Welcome to the world of Python"
print("str[2:10] = ", str[2:10])    # default stride is 1
print("str[2:10:1] = ", str[2:10:1]) # same as stride = 1
print("str[2:10:2] = ", str[2:10:2]) # skips every alternate character
print("str[2:13:4] = ", str[2:13:4]) # skips every fourth character
```

OUTPUT

```
str[2:10] = lcome to
str[2:10:1] = lcome to
str[2:10:2] = loet
str[2:13:4] = le
```

Write a program that asks the user to enter a string. The program should print the following:

- (a) The total number of characters in the string
- (b) The string should repeat 10 times using '*'(multiplication).
- (c) The first character of the string (remember that string indices start at 0)
- (d) The first three characters of the string
- (e) The last three characters of the string
- (f) The string backwards(means reverse)
- (g) The seventh character of the string if the string is long enough and a error message otherwise
- (h) The string with its first and last characters removed(use slice operation)
- (i) The string in all caps
- (j) The string with every 'a' replaced with an 'e'
- (k) The string with every space replace by '-'
- (l) The string with every letter replace by space

(m) Remove a specific character of a string based on index value(use slice operation)

Ans:

#Program on slice operation on strings

```
s=input("Enter a String")
l=len(s)
print("Number of characters in string",s,"is:",l)
print("string",s,"repeated 10 times:",s*10)
print("The first character of the string",s,"is:",s[0])
print("The first three characters of the string",s,"are:",s[0:3])
print("The last three characters of the string",s,"are:",s[-3:])
print("The string",s,"in backward:",s[::-1])
if(l<7):
    print("string",s,"is not enough")
else:
    print("string",s,"s 7th character is: ",s[6])
print("String",s,"after removing first and last characters.",s[1:-1]) #here l is the length of the string
print("String",s,"in all caps:",s.upper())
print("String",s,"after replacing every a by e:",s.replace("a","e"))
os=s #os means original string
for i in s:
    s=s.replace(i," ")
print("String \"",os,"\" after replacing every letter by space:",s)
os=s #os means original string
for i in s:
```

```
print("String \ " ,os,* \ "after replacing every letter by space.",s)
print(input("enter index value of the character to be deleted"))
print("String after deleting the specified character:-",os[:i]+os[i+1:])
```

ord() and chr() Functions :

ord() function returns the ASCII number of the character and chr() function returns character represented by a ASCII number.

ACII (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE)
values of A-Z IS 65-90 ,a-z is 97-122 and '0' to '9' is 48 to 57

Examples:

ch = 'R' print(ord(ch))	print(chr(82))	print(chr(112))	print(ord('p'))
OUTPUT	OUTPUT	OUTPUT	OUTPUT
82	R	p	112

2:

Ch="A"

Ch=chr(ord(Ch)+1) # now the Ch value is B

Programs:

1. Write a Program to display the following pattern

```
A
BB
CCC
DDDD
EEEE
```

2. Write a Program to display the following pattern

```
A
AB
ABC
ABCD
ABCDE
```

in and not in Operators (Membership operators)with strings

in and not in operators can be used with strings to determine whether a string is present in another string. Therefore, the in and not in operator are also known as membership operators.

Examples:

```
str1 = "Welcome to the world of Python  
!!!"  
str2 = "the"  
if str2 in str1:  
    print("Found")  
else:  
    print("Not Found")
```

OUTPUT

Found

```
str1 = "This is a very good book"  
str2 = "best"  
if str2 not in str1:  
    print("The book is very good but it  
may not be the best one.")  
else:  
    print ("It is the best book.")
```

OUTPUT

The book is very good but it may not be
the best one.

Comparing Strings

We can use all comparison or relational operators with strings.

Operator	Description	Example
<code>==</code>	If two strings are equal, it returns True.	<pre>>>> "Abc" == "Abc" True</pre>
<code>!=</code> or <code><></code>	If two strings are not equal, it returns True.	<pre>>>> "Abc" != "Abc" True >>> "abc" <> "Abc" True</pre>
<code>></code>	If the first string is greater than the second, it returns True.	<pre>>>> "abc" > "Abc" True</pre>
<code><</code>	If the second string is greater than the first, it returns True.	<pre>>>> "abc" < "Abc" True</pre>
<code>>=</code>	If the first string is greater than or equal to the second, it returns True.	<pre>>>> "abc" >= "Abc" True</pre>
<code><=</code>	If the second string is greater than or equal to the first, it returns True.	<pre>>>> "Abc" <= "Abc" True</pre>

Iterating String or Looping a string:

String is a sequence type (sequence of characters). You can iterate through the string using for loop.

Examples:

```
str = "Welcome to Python"
for i in str:
    print(i, end=' ')
```

OUTPUT

```
Welcome to Python
```



```
message = " Welcome to Python "  
index = 0  
while index < len(message):  
    letter = message[index]  
    print(letter, end=' ')  
    index += 1
```

OUTPUT

W e l c o m e t o P y t h o n

Programs:

1. Write a program to input a string and check whether it is palindrome string or not

Ans:

```
s=input("enter a string")  
rs=s[::-1] # rs is a reverse string  
if(s==rs):  
    print(s,"is a plaindrome string")  
else:  
    print(s,"is not a palindrome string")
```

Output:

enter a stringmadam

madam is a plaindrome string

2. Write a Program to input a sentence and find number of alphabets, digits and special symbols in it.

```
s=input("enter a sentence")
noa=nod=nos=0
s=input("enter a sentence")
noa=nod=nos=0
for i in s:
    if(i.isalpha()):
        noa=noa+1
    elif(i.isdigit()):
        nod = nod+1
    else:
        nos=nos+1
print("no of alphabets=",noa,"no of digits=",nod,"number of special symbols=",nos)
```

Output:

```
enter a sentencewelcome to python@007
no of alphabets= 15 no of digits= 3 number of special symbols= 3
```

3. Write a Program to input a sentence and find number of vowels in it.

(Hint: for i in "aeiouAEIUOU" :)

The String Module

The string module consist of a number of useful constants, classes and functions (some of which are deprecated). Some constants defined in the string module (string)are:

import string

- 1) string.ascii_letters: it refers both upper case and lower case constants

- 2) `string.ascii_lowercase`: Refers to all lowercase letters from a-z
- 3) `string.ascii_uppercase`: Refers to all uppercase letters from A-Z
- 4) `string.digits`: Refers to digits from 0-9
- 5) `string.printable`: Refers to all digits, letters, special symbols

Examples:

```
print(string.ascii_letters)
```

Output:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Some functions:

```
str = "Welcome to the world of Python"
print("Uppercase - ", str.upper())
print("Lowercase - ", str.lower())
print("Split - ", str.split())
print("Join - ", '-'.join(str.split()))
print("Replace - ", str.replace("Python", "Java"))
print("Count of o - ", str.count('o'))
print("Find of - ", str.find("of"))
```

OUTPUT

```
Uppercase -  WELCOME TO THE WORLD OF PYTHON
Lowercase -  welcome to the world of python
Split -  ['Welcome', 'to', 'the', 'world', 'of', 'Python']
Join -  Welcome-to-the-world-of-Python
Replace -  Welcome to the world of Java
Count of o -  5
Find of -  21
```

Programming Tip: A method is called by appending its name to the variable name using the period as a delimiter.

Data Structure

A *data structure* is a group of data elements that are put together under one name. Data structure defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

Data Structure: Sequence

A sequence is a data type that represents a group of elements. The purpose of any sequence is to store and process group elements. In python, strings, lists, tuples and dictionaries are very important sequence data types.

LIST:

A list is similar to an array that consists of a group of elements or items. Just like an array, a list can store elements. But, there is one major difference between an array and a list. An array can store only one type of elements whereas a list can store different types of elements. Hence lists are more versatile and useful than an array.

Creating a List:

Creating a list is as simple as putting different comma-separated values between square brackets.

```
student=[556, "Mothi", 84, 96, 84, 75, 84]
```

We can create empty list without any elements by simply writing empty square brackets as: `student=[]`

We can create a list by embedding the elements inside a pair of square braces []. The elements in the list should be separated by a comma (,).

Accessing Values in list:

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. To view the elements of a list as a whole, we can simply pass the list name to print function.

negative indexing	-7	-6	-5	-4	-3	-2	-1
positive indexing:	0	1	2	3	4	5	6
Student:	556	"Mothi"	84	96	84	75	84

Ex:

```
student = [556, "Mothi", 84, 96, 84, 75, 84]
print student
print student[0] # Access 0th element
print student[0:2] # Access 0th to 1st elements
print student[2:] # Access 2nd to end of list elements
print student[:3] # Access starting to 2nd elements
print student[-] # Access starting to ending elements
print student[-1] # Access last index value
print student[-1:-7:-1] # Access elements in reverse order
```

Output:

```
[556, "Mothi", 84, 96, 84, 75, 84]
Mothi
[556, "Mothi"]
```

```
[84, 96, 84, 75, 84]
[556, "Mothi", 84]
[556, "Mothi", 84, 96, 84, 75, 84]
84
[84, 75, 84, 96, 84, "Mothi"]
```

Creating lists using range() function:

We can use range() function to generate a sequence of integers which can be stored in a list. To store numbers from 0 to 10 in a list as follows.

```
numbers = list(range(0,11))
print numbers # [0,1,2,3,4,5,6,7,8,9,10]
```

To store even numbers from 0 to 10 in a list as follows.

```
numbers = list(range(0,11,2))
print numbers # [0,2,4,6,8,10]
```

Looping on lists:

We can also display list by using for loop (or) while loop. The len() function useful to know the numbers of elements in the list. while loop retrieves starting from 0th to the last element i.e. n-1

Ex-1:

```
numbers = [1,2,3,4,5]
for i in numbers:
    print i
```

Output:

```
1 2 3 4 5
```

Updating and deleting lists:

Lists are *mutable*. It means we can modify the contents of a list. We can append, update or delete the elements of a list depending upon our requirements.

Appending an element means adding an element at the end of the list. To, append a new element to the list, we should use the append() method.

Example:

```
lst=[1,2,4,5,8,6]
print lst      # [1,2,4,5,8,6]
lst.append(9)
print lst      # [1,2,4,5,8,6,9]
```

Updating an element means changing the value of the element in the list. This can be done by accessing the specific element using indexing or slicing and assigning a new value.

Example:

```
lst=[4,7,6,8,9,3]
print lst      # [4,7,6,8,9,3]
lst[2]=5       # updates 2nd element in the list
print lst      # [4,7,5,8,9,3]
lst[2:5]=10,11,12  # update 2nd element to 4th element in the list
print lst      # [4,7,10,11,12,3]
```


Deleting an element from the list can be done using `del` statement. The `del` statement takes the position number of the element to be deleted.

Example:

```
lst=[5,7,1,8,9,6]
del lst[3]      # delete 3rd element from the list i.e., 8
print lst       # [5,7,1,9,6]
```

If we want to delete entire list, we can give statement like `del lst`.

Concatenation of Two lists:

We can simply use `+` operator on two lists to join them. For example, `„x“` and `„y“` are two lists. If we write `x+y`, the list `„y“` is joined at the end of the list `„x“`.

Example:

```
x=[10,20,32,15,16]
y=[45,18,78,14,86]
print x+y      # [10,20,32,15,16,45,18,78,14,86]
```

Repetition of Lists:

We can repeat the elements of a list `„n“` number of times using `„**“` operator.

```
x=[10,54,87,96,45]
print x*2      # [10,54,87,96,45,10,54,87,96,45]
```

Membership in Lists:

We can check if an element is a member of a list by using `„in“` and `„not in“` operator. If the element is a member of the list, then `„in“` operator returns `True` otherwise returns `False`. If the element is not in the list, then `„not in“` operator returns `True` otherwise returns `False`.

Example:

```
x=[10,20,30,45,55,65]
a=20
print a in x    # True
a=25
print a in x    # False
a=45
print a not in x # False
a=40
print a not in x # True
```

Aliasing and Cloning Lists:

Giving a new name to an existing list is called *‘aliasing’*. The new name is called *‘alias name’*. To provide a new name to this list, we can simply use assignment operator (`=`).

Example:

```
x = [10, 20, 30, 40, 50, 60]
y=x      # x is aliased as y
print x  # [10,20,30,40,50,60]
print y  # [10,20,30,40,50,60]
x[1]=90  # modify 1st element in x
print x  # [10,90,30,40,50,60]
print y  # [10,90,30,40,50,60]
```




In this case we are having only one list of elements but with two different names „x“ and „y“. Here, „x“ is the original name and „y“ is the alias name for the same list. Hence, any modifications done to „x“ will also modify „y“ and vice versa.

Obtaining exact copy of an existing object (or list) is called „cloning“. To Clone a list, we can take help of the slicing operation [:].

Example:

```
x = [10, 20, 30, 40, 50, 60]
y = x[:]          # x is cloned as y
print x           # [10,20,30,40,50,60]
print y           # [10,20,30,40,50,60]
x[1]=90           # modify 1st element in x
print x           # [10,90,30,40,50,60]
print y           # [10,20,30,40,50,60]
```



When we clone a list like this, a separate copy of all the elements is stored into „y“. The lists „x“ and „y“ are independent lists. Hence, any modifications to „x“ will not affect „y“ and vice versa.

Basic List Operations

Method	Description	Syntax	Example	Output
append()	Appends an element to the list. In insert(), if the index is 0, then element is inserted as the first element and if we write, list.insert(len(list), obj), then it inserts obj as the last element in the list. That is, if index= len(list) then insert() method behaves exactly same as append() method	list. append(obj)	num_list = {0,1,2,0,3,2,4,0} num_list. append(10) print (num_list)	{0,3,2, 0,1,2, 4,0,10}
count()	Counts the number of times an element appears in the list.	list.count (obj)	print(num_list. count(4))	1
index()	Returns the lowest index of obj in the list. Gives a ValueError if obj is not present in the list.	list. index(obj)	>>> num_list = {0,1,2,0,3,2,6,0} >>> print(num_ list.index(2))	2

insert()	Inserts obj at the specified index in the list.	list. insert(index, obj)	<pre> >>> num_list = [6,3,7, [6,3,7,0,3,7,6,0] 100,0, >>> num_list. 3,7,6, insert(3, 100) 0] >>> print(num_list) </pre>
pop()	Removes the element at the specified index from the list. Index is an optional parameter. If no index is specified, then removes the last object (or element) from the list.	list. pop([index])	<pre> num_list = 9 [6,3,7,0,1,2,4,9] [6, 3, print(num_list. 7, 0, 1, pop()) 2, 4] print(num_list) </pre>

List Methods:

remove()	Removes or deletes obj from the list. ValueError is generated if obj is not present in the list. If multiple copies of obj exists in the list, then the first value is deleted.	list. remove(obj)	<pre> >>> num_list = [6,3,7, [6,3,7,0,3,2,4,9] 1,2,4, >>> num_list. 9] remove(0) >>> print(num_list) </pre>
reverse()	Reverse the elements in the list	list. reverse()	<pre> >>> num_list = [6,3,7, [6,3,7,0,3,2,4,9] 2,3,7, >>> num_list. 3, 6] reverse() >>> print(num_list) </pre>
sort()	Sorts the elements in the list	list.sort()	<pre> >>> num_list = [6,3,7, [6,3,7,0,3,2,4,9] 2,3,0, >>> num_list. 7, 3, sort() 0] >>> print(num_list) </pre>
extend()	Adds the elements in a list to the end of another list. Using + or += on a list is similar to using extend().	list1. extend(list2)	<pre> >>> num_list1 = [1, 2, [1,2,3,4,5] 3,4,5, >>> num_list2 = [6,7,8, [6,7,8,9,10] 9,10] >>> num_list1. extend(num_list2) >>> print(num_list1) </pre>

More Methods in Lists:

Method	Description
<i>list.index(x)</i>	Returns the first occurrence of x in the list.
<i>list.append(x)</i>	Appends x at the end of the list.
<i>list.insert(i,x)</i>	Inserts x to the list in the position specified by i.
<i>list.copy()</i>	Copies all the list elements into a new list and returns it.
<i>list.extend(list2)</i>	Appends list2 to list.
<i>list.count(x)</i>	Returns number of occurrences of x in the list.
<i>list.remove(x)</i>	Removes x from the list.

<i>lst.pop()</i>	Removes the ending element from the list.
<i>lst.sort()</i>	Sorts the elements of list into ascending order.
<i>lst.reverse()</i>	Reverses the sequence of elements in the list.
<i>lst.clear()</i>	Deletes all elements from the list.
<i>max(lst)</i>	Returns biggest element in the list.
<i>min(lst)</i>	Returns smallest element in the list.

Example:

```
lst=[10,25,45,51,45,51,21,65]
lst.insert(1,46)
print lst      # [10,46,25,45,51,45,51,21,65]
print lst.count(45)  # 2
```

Nested Lists:

A list within another list is called a *nested list*. We know that a list contains several elements. When we take a list as an element in another list, then that list is called a nested list.

Example:

```
a=[10,20,30]
b=[45,65,a]
print b      # display [ 45, 65, [ 10, 20, 30 ] ]
print b[1]   # display 65
print b[2]   # display [ 10, 20, 30 ]
print b[2][0] # display 10
print b[2][1] # display 20
print b[2][2] # display 30
for x in b[2]:
    print x,      # display 10 20 30
```

Nested Lists as Matrices:

Suppose we want to create a matrix with 3 rows 3 columns, we should create a list with 3 other lists as:

```
mat= [ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]
```

Here, „mat“ is a list that contains 3 lists which are rows of the „mat“ list. Each row contains again 3 elements as:

```
[ [ 1, 2, 3 ] ,      # first row
  [ 4, 5, 6 ] ,      # second row
  [ 7, 8, 9 ] ]      # third row
```

Example:

```
Mat=[1,2,3],[4,5,6],[7,8,9]
for r in Mat:
    print(r,end=" ")
    print()
for i in range(3):
    for j in range(3):
        print(Mat[i][j],end=" ")
    print()
```

Output:

```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

```
1 2 3
4 5 6
7 8 9
```

One of the main use of nested lists is that they can be used to represent matrices. A matrix represents a group of elements arranged in several rows and columns. In python, matrices are created as 2D arrays or using matrix object in numpy. We can also create a matrix using nested lists.

Q) Write a program to perform addition of two matrices.

```
a=[[1,2,3],[4,5,6],[7,8,9]]
b=[[4,5,6],[7,8,9],[1,2,3]]
for i in range(3):
    for j in range(3):
        print(a[i][j]+b[i][j],end=" ")
    print ( )
```

Output:

```
5      7      9
11     13     15
8      10     12
```

Q) Write a program to perform multiplication of two matrices.

```
a=[[1,2,3],[4,5,6]]
b=[[4,5],[7,8],[1,2]]
c=[[0,0],[0,0]]
for i in range(2):
    for j in range(2):
        for k in range(3):
            c[i][j] += a[i][k]*b[k][j]
for i in range(0,m1):
    for j in range(0,n2):
        print(c[i][j],end=" ")
    print ( )
```

Output:

```
21     27
57     72
```


List Comprehensions:

List comprehensions represent creation of new lists from an iterable object (like a list, set, tuple, dictionary or range) that satisfy a given condition. List comprehensions contain very compact code usually a single statement that performs the task.

We want to create a list with squares of integers from 1 to 100. We can write code as:

```
squares=[]  
for i in range(1,11):  
    squares.append(i**2)
```

The preceding code will create „squares“ list with the elements as shown below:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

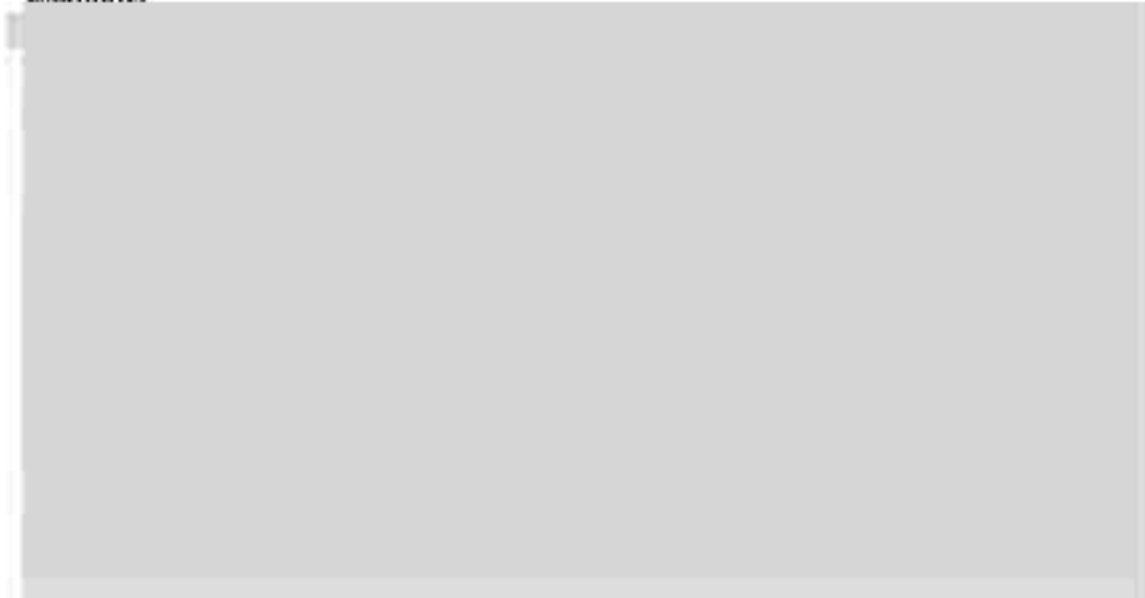
The previous code can be rewritten in a compact way as:

```
squares=[x**2 for x in range(1,11)]
```

This is called list comprehension. From this, we can understand that a list comprehension consists of square braces containing an expression (i.e., $x**2$). After the expression, a `for` loop and then zero or more `if` statements can be written.

```
[ expression for item1 in iterable if statement1  
  for item1 in iterable if statement2  
  for item1 in iterable if statement3 .....]
```

Example:

**Using the enumerate() and range() Functions**

`enumerate()` function is used when you want to print both index as well as an item in the list. The function returns an enumerate object which contains the index and value of all the items of the list as a tuple. The `range()` function is used when you need to print index.

Example:

```
X=[1,2]  
for i,value in enumerate(X):  
    print("Index=","i","value=",value)
```

Output:

Index=0 value=1

Index=1 value=2

PROGRAMS and EXERCISE:

1. Write a program that asks the user to enter a list of integers. Do the following:
 - (a) Print the total number of items in the list.
 - (b) Print the last item in the list.
 - (c) Print the list in reverse order.
 - (d) Print Yes if the list contains a 5 and No otherwise.
 - (e) Print the number of fives in the list.
 - (f) Remove the first and last items from the list, sort the remaining items, and print the result.
 - (g) Print how many integers in the list are less than 5.
 - (h) Print the average of the elements in the list.
 - (i) Print the largest and smallest values in the list.
 - (j) Print the second largest and second smallest entries in the list.
 - (k) Print how many even numbers are in the list.

Ans:

```
#program on lists
list=[]
n=int(input("enter how many values u want to store"))
for i in range(n):
    x=int(input("enter a value"))
    list.append(x)
print(list)
print("The total elements=",n)
print("The last item in the list=",list[-1])
dlist=list[:]
dlist.reverse()
print("The list in reverse order=",dlist)
if list.count(5)!=0:
    print("yes")
else:
    print("no")
dlist.reverse()
del(dlist[0])# remove first element of the list
del(dlist[-1])#remove last element of the list
dlist.sort()
print("The remaining elements in sorted order",dlist)
c=0
for i in range(n):
    if(list[i]<5):
        c=c+1
print("no of <5 in list=",c)
print("average of the list=",sum(list)/n)
list.sort()
print("large=",list[-1],"small=",list[0])
print("second large=",list[-2],"second small=",list[1])
c=0
for i in range(n):
    if(list[i]%2==0):
```



```
c=c+1  
print("no. of evens in list=",c)
```

2. Write a Program to define a list of countries as BRICS member and check whether give country is a member or not.
3. Write a program to create a list of numbers from 1 to 100 and then delete all the even numbers from list and finally display it.
4. Write a Program to input a list and value to be searched in the list. If value is found print its index and count how many times it's repeated.
5. Write a program to remove all duplicates from a list.
6. Write a Program to create a list of 100 numbers and then create two other lists for even numbers and odd numbers separately and print them finally.
7. Write a Program to generate the Fibonacci series, store in a list and print that list.

*****ALL THE VERY BEST*****

TUPLE:

A Tuple is a python sequence which stores a group of elements or items. Tuples are similar to lists but the main difference is tuples are immutable whereas lists are mutable. Once we create a tuple we cannot modify its elements. Hence, we cannot perform operations like `append()`, `extend()`, `insert()`, `remove()`, `pop()` and `clear()` on tuples. Tuples are generally used to store data which should not be modified and retrieve that data on demand.

Creating Tuples:

We can create a tuple by writing elements separated by commas inside parentheses `()`.

The elements can be same data type or different types.

- > To create an empty tuple, we can simply write empty parenthesis, as: `tup=()`
- > To create a tuple with only one element, we can, mention that element in parenthesis and after that a comma is needed. In the absence of comma, python treats the element as ordinary data type.

```
tup = (10)
print tup          # display 10
print (type(tup))  # display <type
                  # "int">
```

```
tup = (10,)
print tup          # display 10
print (type(tup))  # display <type
                  # "tuple">
```

- > To create a tuple with different types of elements:
`tup=(10,20,31.5,"Gudivada")`
- > If we do not mention any brackets and write the elements separating them by comma, then they are taken by default as a tuple.

```
tup= 10, 20, 34, 47
```

- > It is possible to create a tuple from a list. This is done by converting a list into a tuple using tuple function.

```
n=[1,2,3,4]
tp=tuple(n)
print tp      # display (1,2,3,4)
```

Another way to create a tuple by using `range()` function that

```
returns a sequence. t=tuple(range(2,11,2))
print t        # display (2,4,6,8,10)
```

Accessing the tuple elements:

Accessing the elements from a tuple can be done using indexing or slicing. This is same as that of a list. Indexing represents the position number of the element in the tuple. The position starts from 0.

```
tup=(50,60,70,80,90)
print( tup[0])      # display 50
print( tup[1:4])    # display (60,70,80)
```

```

print( tup[-1])    # display 90
print( tup[-1:-4:-1] )
#display(90,80,70)
print (tup[-4:-1])
# display (60,70,80)

```

Updating and deleting elements

Tuples are immutable which means you cannot update, change or delete the values of tuple elements.

Example-1:

```

A={1,2,3}
A[1]=33
print(a)
Output:
TypeError

```

Example-2:

```

A={1,2,3}
del(A[1])
print(a)
Output:
TypeError

```

However, you can always delete the entire tuple by using the statement.

```

C:\Python27\Scripts>python tup1.py
a=(1,2,3,4,5)
del a
print a

Traceback (most recent call last):
  File "C:/Python27/tup1.py", line 3, in <module>
    print a
NameError: name 'a' is not defined
>>>

```

Note that this exception is raised because you are trying print the deleted element.

Operations on tuple:

Operation	Description	Example	O/P
len(t)	Return the length of tuple.	t1=(1,2,4,3) t2=(1,"abc",2.6)	4

		<code>print("length of t1=",len(t1))</code>	
<code>tup1+tup2</code>	Concatenation of two tuples.	<code>print("concatnation of t1,t2:",t1+t2)</code>	concatnation of t1,t2: (1, 2, 4, 3, 1, 'abc', 2.6)
<code>tup*n</code>	Repetition of tuple values in n number of times.	<code>print("Repetition of t2:",t2*2)</code>	Repetition of t2: (1, 'abc', 2.6, 1, 'abc', 2.6)
<code>x in tup</code>	Return True if x is found in tuple otherwise returns False.	<code>X=9 X in t1</code>	False
<code>max(tup)</code>	Returns the maximum value in tuple.	<code>print("maximum of t1:",max(t1))</code>	maximum of t1: 4
<code>min(tup)</code>	Returns the minimum value in tuple.	<code>print("minimum of t1:",min(t1))</code>	minimum of t1: 1
<code>tuple(list)</code>	Convert list into tuple.	<code>tuple([1,2,3])</code>	(1,2,3)
<code>tup.count(x)</code>	Returns how many times the element "x" is found in tuple.	<code>print("count of 2 in t1:",t1.count(2))</code>	count of 2 in t1: 1
<code>tup.index(x)</code>	Returns the first occurrence of the element "x" in tuple. Raises ValueError if "x" is not found in the tuple.	<code>print("index of 9 in t2:",t2.index(2.6))</code>	index of 9 in t2: 2
<code>sorted(tup)</code>	Sorts the elements of tuple into ascending order. <code>sorted(tup,reverse=True)</code> will sort in reverse order.	<code>print("sorted of t1 in order:",sorted(t1)) print(t1)</code>	sorted of t1 in order: [1, 2, 3, 4] [1,2,4,3]

Example:

```

tuple1 = (123, 'xyz')
tuple2 = (456, 'abc')
print cmp(tuple1, tuple2)           #display False
print cmp(tuple2, tuple1)           #display False

```

Nested Tuples:

Python allows you to define a tuple inside another tuple. This is called a *nested tuple*.

```

students=(('RAVI', 'CSE', 92.00), ('RAMU', 'ECE', 93.00), ('RAJA', 'EEE', 87.00))
for i in students:
    print(i)

```

Output: ('RAVI', 'CSE', 92.00)
('RAMU', 'ECE', 93.00)
('RAJA', 'EEE', 87.00)

The zip() Function(2 marks question)

The zip() is a built-in function that takes two or more sequences and "zips" them into a list of tuples.

Example: Program to show the use of zip() function

```
Tup = (1,2,3,4,5)
List1 = ['a','b','c','d','e']
print(list(zip(Tup, List1)))
```

OUTPUT

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

Lab Program:

Write a python program to demonstrate various operations on tuples.

```
#various operations on tuples
t1=(1,2,4,3)
t2=(1,'abc',2.6)
print("length of t1:",len(t1))
print("concatnation of t1,t2:",t1+t2)
print("Repetition of t2:",t2*2)
print("maximum of t1:",max(t1))
print("minimum of t1:",min(t1))
print("count of 2 in t1:",t1.count(2))
print("index of 9 in t2:",t2.index(2.6))
print("sorted of t1 in order:",sorted(t1))
```

Output:

```
length of t1= 4
concatnation of t1,t2: (1, 2, 4, 3, 1, 'abc', 2.6)
Repetition of t2: (1, 'abc', 2.6, 1, 'abc', 2.6)
maximum of t1: 4
minimum of t1: 1
count of 2 in t1: 1
index of 9 in t2: 2
sorted of t1 in order: [1, 2, 3, 4]
```

2. Write a Program to find the second largest and smallest number in a tuple

```
T=(3,2,4,1,5,0)
```

```
C=sorted(T)
```

```
print("second largest=",C[-2],"second smallest=",C[0])
```

Advantages of Tuple over List

- **Tuples** are used to store values of *different data types*. **Lists** can however, store data of *similar data types*.
- Since **tuples are immutable**, iterating through tuples is faster than iterating over a list. This means that a tuple performs better than a list(**List is mutable**).
- **Tuples** can be used as **key for a dictionary** but **lists cannot** be used as keys.
- **Tuples** are best suited for storing data that is write-protected(means **only reading not for modification**).
- **Tuples** can be used in place of lists where the number of values is known and **small**.

SET:

Set is another data structure supported by python. Basically, sets are same as lists but with a difference that sets are lists with **no duplicate entries**.

Technically a **set is a mutable and an unordered collection of items**. This means that we can easily add or remove items from it.

Creating a Set:

- > A set is created by placing all the elements inside curly brackets
& Separated by comma or by using the built-in function `set()`.

Syntax:

```
Set_variable_name={var1, var2, var3, var4, .....}
```

Example:

```
s={1, 2.5, "abc"}  
print(s) # display {1, 2.5, 'abc'}
```

- > Creating an empty is possibly through only `set()` function.

Example:

```
S=set() # S is empty set  
print(S) # displays set()
```

- > Even if we write duplicates in the set, but set does not contain duplicates.

Example:

```
S={1,2,1,2,3}  
print(S) # {1,2,3}
```

Converting a list into set:

A set can have any number of items and they may be of different data types. `set()`

function is used to converting list into set.

```
s=set([1, 2.5, "abc"])  
print s # display {1, 2.5, 'abc'}
```

We can also convert tuple or string into set.

```
tup=(1, 2, 3, 4, 5)  
print( set(tup)) # {1, 2, 3, 4, 5}  
str="Vihari"  
print( set(str) ) # display { 'V', 'h', 'i', 'a', 'r' }
```

Note: Set never preserves order

Operations on set:

Some operations on set

S.no	Operation	Description	Example	Output
1	<code>len(A)</code>	number of elements in set A(cardinality)	A={1,2} B={2,3,4,5} <code>len(A)</code>	2
2	<code>x in A</code>	test x for membership in A	X=9 <code>X in A</code>	False

3	x not in A	test x for non-membership in A	X not in A	True
4	maximum	Finds the maximum value of the set	max(A)	2
5	minimum	Finds the minimum value of the set	min(A)	1
6	A.issubset(B) (or) A <= B	test whether every element in A is in B or not	A.issubset(B)	False
7	A.issuperset(B)	test whether every element in B is in A or not	A.issuperset(B)	False
8	A.union(B)	new set with elements from both A and B	A.union(B)	{1,2,3,4,5}
9	A.intersection(B)	new set with common elements from both A and B	A.intersection(B)	{2}
10	A.difference(B)	new set with elements in A but not in B	A.difference(B)	{1}
11	A.symmetric_difference(B)	new set with elements from both A and B but not common elements	A.symmetric_difference(B)	{1,3,4,5}
12	A.add(x)	Adds the element x into set	A.add(9)	{1,2,9}
13	A.remove(x)	Deletes the element x from set if it's present otherwise error.	A.remove(9)	{1,2}

Note:

To create an empty set you cannot write `s={}`, because python will make this as a dictionary. Therefore, to create an empty set use

```
set() function. s=set()          s={}
print(type(s))                   # display <type "set"> print (type(s))#
display <type "dict">
```

Updating a set:

Since sets are unordered, indexing has no meaning. Set operations do not allow users to access or change an element using indexing or slicing.

Example:

```
S={1,"a",2.6,"abc"}
```

```
for I in S:
```

```
print(I,end="")    #1aabc2.6(unordered output)
```

Lab Program:

#various operations on Sets

```
A={1,2}
B={2,3,4,5}
print("A Set=",A)
print("B Set=",B)
print("length of A=",len(A))
print("maximum of B:",max(B))
print("minimum of A:",min(A))
print("A union B =",A.union(B))
print("A intersection B=",A.intersection(B))
print("A difference B=",A.difference(B))
print("A symmetric difference B=",A.symmetric_difference(B))
print("A is subset of B:",A.issubset(B))
print("A is superset of B:",A.issuperset(B))
A.add(9)
print("After Adding new element 9 to set A:",A)
B.remove(4)
print("After Deleting 4 from set B",B)
```

Output:

```
A Set= {1, 2}
B Set= {2, 3, 4, 5}
length of A= 2
maximum of B: 5
minimum of A: 1
A union B = {1, 2, 3, 4, 5}
A intersection B= {2}
A difference B= {1}
A symmetric difference B= {1, 3, 4, 5}
A is subset of B: False
A is superset of B: False
After Adding new element 9 to set A: {1, 2, 9}
After Deleting 4 from set B {2, 3, 5}
```


Dictionary:

A dictionary represents a group of elements arranged in the form of key-value pairs. The first element is considered as "key" and the immediate next element is taken as its "value". The key and its value are separated by a colon (:). All the key-value pairs in a dictionary are inserted in curly braces {}. **Dictionary is mutable by nature.**

```
d= {"Regd.No": 556, "Name": "Kothi", "Branch": "CSE" }
```

Here, the name of dictionary is "d". The first element in the dictionary is a string "Regd.No". So, this is called "**key**". The second element is 556 which is taken as its "**value**".

Example:

```
d= {"Regd.No": 556, "Name": "Kothi", "Branch": "CSE" }  
print (d["Regd.No"])      # 556  
print (d["Name"])         # Kothi  
print (d["Branch"])       # CSE
```

- To access the elements of a dictionary, we should not use indexing or slicing.
- For example, `d[0]` or `d[1:3]` etc. expressions will give error.
- To access the value associated with a key, we can mention the key name inside the square braces, as: `d["Name"]`.

Example:

```
d= {"Regd.No": 556, "Name": "Kothi", "Branch": "CSE" }  
print (d["Regd.No"])      # 556  
print (d["Name"])         # Kothi  
print (d["Branch"])       # CSE
```

- If we want to know how many key-value pairs are there in a dictionary, we can use the `len()` function, as shown

```
d= {"Regd.No": 556, "Name": "Kothi", "Branch":  
"CSE"} print( len(d))    # 3
```

- We can also insert a new key-value pair into an existing dictionary. This is done by mentioning the key and assigning a value to it.

```
d={"Regd.No": 556, "Name": "Kothi", "Branch": "CSE"}  
print (d)  #{'Branch': 'CSE', 'Name': 'Kothi',  
Regd.No': 556} d["Gender"]="Male"  
print (d)  #{'Gender': 'Male', 'Branch': 'CSE', 'Name': 'Kothi',  
Regd.No': 556}
```

- Suppose, we want to delete a key-value pair from the dictionary, we can use `del` statement as:

```
del d["Regd.No"] #{'Gender': 'Male', 'Branch': 'CSE', 'Name': 'Kothi'}
```

- To Test whether a "key" is available in a dictionary or not, we can use "in" and "not in" operators. These operators return either True or False.

`"Name" in d` #check if "Name" is a key in d and returns True/

- > We can use **any data types for value**.

For example, a value can be a number, string, list, tuple or another dictionary.

- > But keys should obey the rules:

- > **Keys should be unique.** It means, *duplicate keys are not allowed. If we enter same key again, the old key will be overwritten and only the new key will be available.*

```
emp={'nag':10,"Vishnu":20,"nag":20}
print emp # {'nag': 20, 'vishnu': 20}
```

- > **Keys should be immutable type.**

For example, we can use a number, string or tuples as keys since they are immutable. We cannot use lists or dictionaries as keys. If they are used as keys, we will get **"TypeError"**.

```
emp={['nag']:10,'vishnu':20,'nag':20}
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    emp={['nag']:10,'vishnu':20,'nag':20}
TypeError: unhashable type: 'list'
```

Dictionary Methods:

Method	Description	Example
<code>d.clear()</code>	Removes all key-value pairs from dictionary "d".	<pre>dict={"Regd.No": 556, "Name": "Kothi", "Branch": "CSE"} d={"Regd.No": 556, "Name": "Kothi", "Branch": "CSE"} d.clear() # {}</pre>
<code>d2=d.copy()</code>	Copies all elements from "d" into a new dictionary d2.	<pre>c=dict.copy() print(c) #{'Branch': 'CSE', 'Name': 'Kothi', 'Regd.No': 556}</pre>
<code>d.get(key)</code>	Returns the value associated with key "k". If key is not found, it returns None/Nothing.	<pre>d.get("Name") # 'Kothi'</pre>
<code>d.items()</code>	Returns an object that contains key-value pairs of "d". The pairs are stored as tuples in the object.	<pre>dict.items() #{'Branch': 'CSE', 'Name': 'Kothi', 'Regd.No': 556}</pre>
<code>d.keys()</code>	Returns a sequence of keys from the dictionary "d".	<pre>dict.keys() # dict_keys(['Regd.No', 'Name', 'Branch'])</pre>
<code>d.values()</code>	Returns a sequence of values from the dictionary "d".	<pre>dict.values() # dict_values([556, 'Kothi', 'CSE'])</pre>
<code>d.update(x)</code>	Adds all elements from dictionary "x" to "d".	<pre>c={"Gender": "Male"} dict.update(c) #{'Regd.No': 556, 'Name': 'Kothi', 'Branch': 'CSE', 'Gender': 'Male'}</pre>

<code>d.pop(key)</code>	Removes the key "k" and its value from "d" and returns the value. If key is not found then "KeyError" is raised.	<code>dict.pop("Name")</code> # 'Kohli' <code>print(d)</code> # {'Regd.No': 556, 'Branch': 'CSE', 'Gender': 'Male'}
<code>len()</code>	Gives how many key-value pairs are there in the dictionary	<code>len(dict)</code> #3

Using for loop with Dictionaries:

for loop is very convenient to retrieve the elements of a dictionary. Let's take a simple dictionary that contains color code and its name as:

```
colors = {'r': 'RED', 'g': 'GREEN', 'b': 'BLUE', 'w': 'WHITE'}
```

Here, "r", "g", "b", "w" represents keys and "RED", "GREEN", "BLUE" and "WHITE" indicate values.

```
colors = {'r': 'RED', 'g': 'GREEN', 'b': 'BLUE', 'w': 'WHITE'}
for k in colors:
    print(k) # displays only
keys for k in colors:
    print(colors[k]) # keys to dictionary and display the values
```

Nested dictionaries:

Dictionary contains another dictionary within it called **Nested Dictionary**.

Example: `d={'Raj':{'Maths':90,'PPG':97,'Acy':87,'BEE':100}}`

Q) Lab program 1

Write a python Program to demonstrate various dictionary operations

```
d= {'Regd.No': 556, 'Name': 'Kohli', 'Branch': 'CSE'}
print ("d['Regd.No']=" ,d['Regd.No'])
print ("d['Name']=" ,d['Name'])
print ("d['Branch']=" ,d['Branch'])
print("Number of key value pairs in dictionary d=", len(d))
d['Gender']="Male"
print ("dictionary d=",d)
del(d['Regd.No'])
print("dictionary d=",d)
print("\'Name\' in d=", 'Name' in d)
c=d.copy()
print("After copying dictionary d to c=",c)
```



```

print("value of key Name=",d.get("Name"))
print("Items of dictionary d=",d.items())
print("Keys of dictionary d=",d.keys() )
print("Values of dictionary d=",d.values() )
c={"Marks":94}
d.update(c)
print("After Updating dictionary d=",d)

```

Output:

```

d["Regd.No"] = 556
d["Name"] = Kohli
d["Branch"] = CSE
Number of key value pairs in dictionary d = 3
dictionary d = {'Regd.No': 556, 'Name': 'Kohli', 'Branch': 'CSE',
'Gender': 'Male'}
dictionary d = {'Name': 'Kohli', 'Branch': 'CSE', 'Gender': 'Male'}
'Name' in d = True
After copying dictionary d to c = {'Name': 'Kohli', 'Branch': 'CSE',
'Gender': 'Male'}
value of key Name = Kohli
Items of dictionary d = dict_items([('Name', 'Kohli'), ('Branch',
'CSE'), ('Gender', 'Male')])
Keys of dictionary d = dict_keys(['Name', 'Branch', 'Gender'])
Values of dictionary d = dict_values(['Kohli', 'CSE', 'Male'])
After Updating dictionary d = {'Name': 'Kohli', 'Branch': 'CSE', 'Gender': 'Male',
'Marks': 94}

```

Lab Program 2:

Q) Write a program that uses a dictionary that contains user names and passwords create by the user. The program should read the user input to enter their username and password. If the username is not in the dictionary, the program should indicate that the person is 'Not a valid user'. If the username is in the dictionary, but the user does not enter the right password, the program should say that the 'Password is invalid'

```

#lab program on dictionaries based on user input
n=int(input("how many user names u want"))
d={}
for i in range(n):
    username=input("enter user name")
    password=input("enter password")
    d[username]=password
print(d)
username=input("enter user name")
if(username in d.keys()):
    password=input("enter password")

```

```

if(d[username]==password):
    print("successfully logged")
else:
    print("invalid password")
else:
    print("invalid username")

```

Output:

```

how many user names u want3
enter user namereyan
enter password20915
enter user namevihan
enter password182017
enter user nameranjith
enter password3282
{reyan: '20915', vihan: '182017', ranjith: '3282'}
enter user namereyan
enter passwordabc
invalid password

```

Q) A Python program to create a dictionary and find the sum of values.

```

d={'m1':85,'m2':84,'eng':86,'c':91}
sum=0
for i in d.values():
    sum+=i
print (sum)    # 346

```

Q) A Python program to create a dictionary with cricket player's names and scores in a match. Also we are retrieving runs by entering the player's name.

```

n=input("Enter How many players? ")
d={}
for i in range(n):
    k=input("Enter Player name: ")
    v=input("Enter score: ")
    d[k]=v
print (d)
name=input("Enter name of player for score: ")
print ("The Score is",d[name])

```

Output:

```

Enter How many
players? 3
Enter Player name:
Sachin
Enter score: 98
Enter Player name:
Sehwag
Enter score: 91
Enter Player name:
Dhoni
Enter score: 95
{'Sehwag': 91, 'Sachin': 98,
'Dhoni': 95}
Enter name of player for score:
Sehwag
The Score is 91

```

Q) Write a program that uses a dictionary that contains ten user names and passwords. The program should read the user input to enter their username and password. If the username is not in the dictionary, the program should indicate that the person is 'Not a valid user'. If the username is in the dictionary, but the user does not enter the right password, the program should say that the 'Password is invalid'

```

#lab program 20 on dictionaries
d={561:"a",562:"b",563:"c",564:"d",565:"e",566:"f",567:"g",568:"h",569:"i",570:"j"}
username=input("enter user name")
if(username in d.keys()):
    password=input("enter password")
    if(password in d.values()):
        print("successfully logged")
    else:
        print("password is invalid")
else:
    print("Not a valid user")

```

Output:

```

enter user name563
enter passwordc
successfully logged

```

Difference between a List and a Dictionary

- First, a list is an ordered set of items. But, a dictionary is a data structure that is used for matching one item(key) with another (value).

- Second, in lists, you can use indexing to access a particular item. But, these indexes should be a number. In dictionaries, you can use any type (immutable) of value as an index.
For example, when we write Dict['Name'], Name acts as an index but it is not a number but a string.
- Third, lists are used to look up a value whereas a dictionary is used to take one value and look up another value. For this reason, dictionary is also known as a *lookup table*.
- Fourth, the key-value pair may not be displayed in the order in which it was specified while defining the dictionary.

Additional Topics:

Converting Lists into Dictionary:

When we have two lists, it is possible to convert them into a dictionary. For example, we have two lists containing names of countries and names of their capital cities.

There are two steps involved to convert the lists into a dictionary. The first step is to create a "zip" class object by passing the two lists to zip() function. The zip() function is useful to convert the sequences into a zip class object. The second step is to convert the zip object into a dictionary by using dict() function.

Example:

```
countries = [ 'USA', 'INDIA', 'GERMANY', 'FRANCE' ]
cities = [ 'Washington', 'New Delhi', 'Berlin', 'Paris' ]
z=zip(countries, cities)
d=dict(z)
print d
```

Output:

```
{'GERMANY': 'Berlin', 'INDIA': 'New Delhi', 'USA': 'Washington', 'FRANCE': 'Paris'}
```

Converting Strings into Dictionary:

When a string is given with key and value pairs separated by some delimiter like a comma (,) we can convert the string into a dictionary and use it as dictionary.

```
s="Vijay-23,Ganesh-20,Lakshmi-19,Nikhil-22" s1=s.split(',')
s2=[]
d={}
for i in s1:
```



```

s2.append(i.split(
    '='))
print d
{'Ganesh': '20', 'Lakshmi': '19', 'Nikhil': '22', 'Vijay': '23'}

```

STRING (VS) LIST (VS) TUPLE (VS) SET (VS) DICTIONARY

Datatype/ Data structure	description	Is it mutable or immutable?	Insertion order preserved or not?	Indexing/slicing	duplicates	Functions/operators
string	Sequence of characters Ex: s="ppg"	Immutable	Not applicable	yes	Not applicable	1) len() 2) max() 3) min() 4) membership operators(in, not in) 5) concatenation(+) 6) repetition(*) 7) count(x) 8) index(x) 9) isalpha() 10) isdigit() 11) isalnum() 12) isupper() 13) islower() 14) isspace() 15) find(x) 16) upper() 17) lower() 18) split() 19) join() 20) reverse()
List	Ordered collection of elements Ex: L=[1,2,3]	mutable	Yes	yes	Yes	1) len() 2) max() 3) min() 4) count(x) 5) index(x) 6) concatenation(+) 7) repetition(*) 8) membership operators(in, not in) 9) append(x)

						10)remove(x) 11)copy() 12)reverse() 13) insert(index,value) 14)sort()
tuple	Read only version of list Ex:t=(1,2,3)	Immutable	Yes	yes	Yes	1) len() 2)max() 3)min() 4) membership operators(in,not in) 5) concatenation(+) 6)repetition(*) 7)count(x) 8)index(x) 9)sorted(tuple) 10)converting list to tuple tuple([1,2,3])
Set	Unordered collection of unique elements Ex: Set={1,2,3}	mutable	no	No	No	1)len() 2)max() 3)min() 4)membership operators(in,not in) 5)union() 6)intersection() 7)difference() 8)symmetric_difference() 9)issubset() 10)issuperset() 11)add() 12)remove() 13)del()
dictionary	Collection key-value pairs Ex: D={1:"a",2:"b"}	mutable	no	no	No duplicate key but values can be duplicated.	1) len() 2)membership operators(in,not in) 3)copy() 4)clear() 5)get(key) 6)keys() 7)values() 8)items() 9)update(x) 10)del()

--	--	--	--	--	--	--

Note:

We can add the following points also

- 1) Creating a data structure syntax
- 2) Creating an empty data structure
- 3) Give example for accessing the elements of a data type or data structure.
- 4) Give example for some functions(minimum 8)

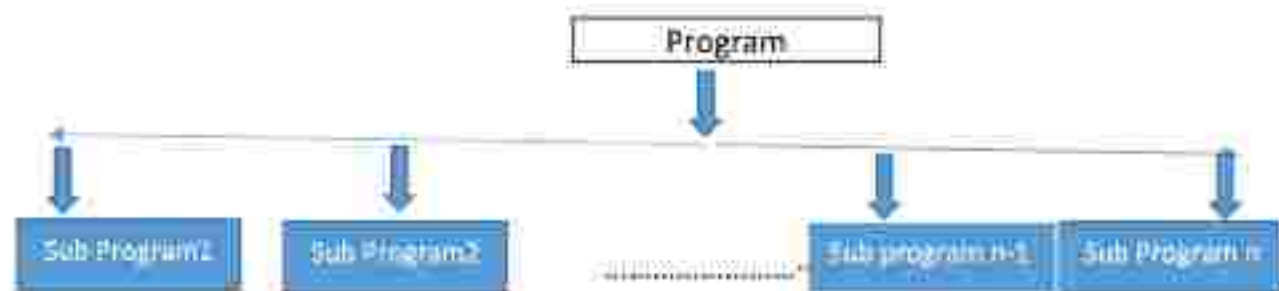
FUNCTIONS

So far we have written programs to solve small problems. When we want to solve large or complex problems, our program may contain more statements means the length of the program (size of the program) increases. This causes the following drawbacks

Drawbacks or Disadvantages:

- Program size is big (number of statements are more).
- Understanding of the program logic is difficult.
- Identifying the mistakes in the program also difficult.
- Same code of the program (statements) may be repeated.

To overcome the above drawbacks, instead of writing a big program for large or complex problems. We divide the actual program into sub programs. The process of dividing the program in to subprograms is called **modularization** and this approach is called **Top down approach**.



SUB PROGRAM OR FUNCTION:

Sub program is called as **Function or Module**.

Function: it contains group of statements to solve a problem. Functions are **two types**

- Standard functions
- User defined functions.

Standard Functions: These functions are already available in the header files. These functions are also called library functions, system defined functions or built in functions.

Examples:

- `print()` `input()` -these are standard output and input functions

User defined functions: These functions are created or defined by the user to solve a problem. User can create or define own functions to solve the problems.

Examples:

- `Myfunction()` - user can create a function like `Myfunction()`.

Defining a Function or user defined function:

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses `()`.
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon `:` and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return none`.

Syntax:

```
def functionname (parameters):
```

```
    """function_docstring"""
```

```
    function_suite
```

```
    return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example:

```
def add(a,b):  
    """This function sum the numbers"""  
    c=a+b  
    print c  
    return
```

Here, "*def*" represents starting of function. "*add*" is function name. After this name, parentheses () are compulsory as they denote that it is a function and not a variable or something else. In the parentheses we wrote two variables "*a*" and "*b*" these variables are called "parameters". A parameter is a variable that receives data from outside a function. So, this function receives two values

from outside and those are stored in the variables "*a*" and "*b*". After parentheses, we put colon (:) that represents the beginning of the function body. The function body contains a group of statements called "suite".

Calling function and Called function:

The function which calls another function is called as ***calling function***.

The function which is called by calling function is called as ***Called function***.

Example:

```
def f( ):  
  
    _____  
  
    g()  
  
    _____  
  
def g( ):  
  
    _____  
  
    _____  
  
    _____  
  
    _____
```


Here f() is called as calling function and g() is called as called function.

Function call

Function gets invoked when it is called. To use the function we have to call that function. We can call the function many ways. Some ways are:

1. Using function name.
Example: add ()
2. using function name with parameters (only variables or constants)
Example: add (5, 6, 7)
Sub (a, b)
3. function with parameters and return value
Ex: result=fact(n)

Actual arguments and Formal Arguments

The arguments that are specified at calling a function are called Actual arguments and the arguments that are specified at called function are called formal arguments. Arguments are called Parameters.

Example:

```
def f(): # calling function
    ____
    ____
    G(p) # calling function G()
    ____

def G(int A):# called function
    ____
    ____
```

In the Above example f () is called as calling function G () is called as called function and p is called as actual argument and A is called as formal argument.

Return statement

In python language, every function returns a value by using return statement. return statement available in various forms.

- return

It means return statement is not returning any value.(it returns nothing).

- Syntax:
- return expression

Expression may contain a variable, constant or both

Example:

```
return 9
return 1.7
return 'n' # it is a string
return p #here p can be any type of variable.
return 5, 2, 7, 3 #we can return all the 4 values at a time
return 9-5+6 # here first the expression will be solved then
that value will be returned.
return a+b
return a-9+b*c/d*4
```

The actual arguments used in a function call are of 4 types:

- a) Positional arguments
- b) Keyword arguments
- c) Default arguments
- d) Variable length arguments

a) Positional Arguments:

These are the arguments passed to a function in correct positional order. Here, the number of arguments and their position in the function definition should match exactly with the number and position of argument in the function call.

```
def attach(s1,s2):  
    s3=s1+s2  
    print s3  
  
attach("New","Delhi") #Positional arguments
```

This function expects two strings that too in that order only. Let's assume that this function attaches the two strings as `s1+s2`. So, while calling this function, we are supposed to pass only two strings as: **`attach("New","Delhi")`**.

The preceding statements displays the following output NewDelhi. Suppose, we passed "Delhi" first and then "New", then the result will be: "DelhiNew". Also, if we try to pass more than or less than 2 strings, there will be an error.

b) Keyword Arguments:

Keyword arguments are arguments that identify the parameters by their names. For example, the definition of a function that displays grocery item and its price can be written as:

`def grocery(item, price):`

At the time of calling this function, we have to pass two values and we can mention which value is for what. For example,

`grocery(item='sugar', price=50.75)`

Here, we are mentioning a keyword „item“ and its value and then another keyword „price“ and its value. Please observe these keywords are nothing but the parameter names which receive these values. We can change the order of the arguments as:

`grocery(price=88.00, item='oil')`

In this way, even though we change the order of the arguments, there will not be any problem as the parameter names will guide where to store that value.

```
def grocery(item,price):  
    print( "item=",item )  
    print( "price=",price )
```

```
grocery(item="sugar",price=50.75) # keyword arguments
```

```
grocery(price=88.00,item="oil") # keyword arguments
```

output:

```
item= sugar price= 50.75
```

```
item= oil price= 88.0
```

c) Default Arguments:

We can mention some default value for the function parameters in the definition. Let's take the definition of `grocery()` function as:

```
def grocery(item, price=40.00)
```

Here, the first argument is "item" whose default value is not mentioned. But the second argument is "price" and its default value is mentioned to be 40.00. At the time of calling this function, if we do not pass "price" value, then the default value of 40.00 is taken.

If we mention the "price" value, then that mentioned value is utilized. So, a default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

Example:

```
def grocery(item,price=40.00):  
    print "item=",item  
    print "price=",price  
grocery(item="sugar",price=50.75)  
grocery(item="oil")
```

Output:

```
item= sugar
```

```
price= 50.75
```

```
item= oil
```

```
price= 40.00
```

d) Variable Length Arguments:

Sometimes, the programmer does not know how many values a function may receive. In that case, the programmer cannot decide how many arguments to be given in the function definition. for example, if the programmer is writing a function

add(a,b)

But, the user who is using this function may want to use this function to find sum of three numbers. In that case, there is a chance that the user may provide 3 arguments to this function as:

add(10,15,20)

Then the add() function will fail and error will be displayed. If the programmer want to develop a function that can accept "n" arguments, that is also possible in python. For this purpose, a variable length argument is used in the function definition. a variable length argument is an argument that can accept any number of values. The variable length argument is written with a "*" symbol before it in the function definition as:

def add(farg, *args):

Here, "farg" is the formal; argument and "*args" represents variable length argument. We can pass 1 or more values to this "*args" and it will store them all in a tuple.

Example:

```
def add(farg,*args):
    sum=0
    for i in args:
        sum=sum+i
    print "sum is",sum+farg
```

```
add(5,10)
add(5,10,20)
add(5,10,20,30)
```

Output:

```
Sum is 15
Sum is 35
Sum is 65
```

Local and Global Variables:

When we declare a variable inside a function, it becomes a local variable. A local variable is a variable whose scope is limited only to that function where it is created. That means the local variable value is available only in that function and not outside of that function.

When the variable "a" is declared inside myfunction() and hence it is available inside that function.

PYTHON PROGRAMMING

Once we come out of the function, the variable „a“ is removed from memory and it is not available.

Example-1:

```
def myfunction():  
    a=10  
    print "Inside function",a #display 10  
  
myfunction()  
print "outside function",a # Error, not available
```

Output:

Inside function 10 outside function

NameError: name 'a' is not defined

When a variable is declared above a function, it becomes global variable. Such variables are available to all the functions which are written after it.

Example-2:

```
a=11  
def myfunction():  
    b=10  
    print "Inside function",a #display global  
var  
    print "Inside function",b #display local var  
myfunction()  
print "outside function",a # available  
print "outside function",b # error
```

Output:

Inside function 11

Inside function 10

outside function 11 outside function

NameError: name 'b' is not defined

The Global Keyword:

Sometimes, the global variable and the local variable may have the same name. In that case, the function, by default, refers to the local variable and ignores the global variable. So, the global variable is not accessible inside the function but outside of it, it is accessible.

Sometimes, the global variable and the local variable may have the same name. In

PYTHON PROGRAMMING

that case, the function, by default, refers to the local variable and ignores the global variable. So, the global variable is not accessible inside the function but outside of it, it is accessible.

Example-1:

```
a=11
def myfunction():
    a=10
    print "Inside function",a # display local variable
myfunction()
print "outside function",a # display global variable
```

Output:

```
Inside function 10
outside function 11
```

When the programmer wants to use the global variable inside a function, he can use the keyword "global" before the variable in the beginning of the function body as:

global a

Example-2:

```
a=11
def myfunction():
    global a
    a=10
    print "Inside function",a # display global variable
myfunction()
print "outside function",a # display global variable
```

Output:

```
Inside function 10
outside function 10
```

Lambda Functions Or Anonymous Functions

Lambda or anonymous functions are so called because they are not declared as other functions using the def keyword.

Lambda functions contain only a single line. Its syntax can be given as:

```
lambda arguments: expression
```


Example:

```
sum = lambda x, y: x + y
print("Sum = ", sum(3, 5))
```

OUTPUT

Sum = 8

Programs using Functions:

1. Write a Program to find the factorial of a given number using function

```
def fact(n):
    if(n==0):
        return 1
    else:
        f=1
        for i in range(1,n+1):
            f=f*i
        return f
```

```
n=int(input("enter a number"))
r=fact(n)
print("factorial of",n,"is",r)
```

2. Write a Program to find the reverse of a given number using function

```
def reverse(n):
    rev=0
    while(n!=0):
        d=n%10
        rev=rev*10+d
        n=n//10
    return rev
```

```
n=int(input("enter a number"))
r=reverse(n)
print("reverse number of",n,"is",r)
```

3. Write a Program to find the sum of the digits of a given number using function

```
def sod(n):  
    sum=0  
    while(n!=0):  
        d=n%10  
        sum=sum+d  
        n=n//10  
    return sum  
  
n=int(input("enter a number"))  
r=sod(n)  
  
print("sum of the digits of the number of",n,"is",r)
```

4. Write a Program to find the nth term of a Fibonacci series using function

```
def fibonacci(n):  
    if(n==1):  
        return 0  
  
    elif(n==2):  
        return 1  
  
    else:  
        a=0  
        b=1  
        for i in range(3,n+1):  
            c=a+b  
            a=b  
            b=c  
        return c  
  
n=int(input("enter a number"))  
r= fibonacci(n)  
print(n,"Fibonacci number is",r)
```

Recursive Functions

A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.

Every recursive solution has two major cases, which are as follows:

base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.

• **recursive case**, in which first the problem at hand is divided into simpler sub parts.

1. Write a Program to find the factorial of a given number using recursive function or recursion

#finding the factorial of a number using recursive function

```
def rfact(n):  
    if(n==0):  
        return 1  
    else:  
        return n*rfact(n-1)
```

```
n=int(input("enter a positive number"))  
r=rfact(n)  
print("factorial of",n,"is",r)
```

2. Write a Program to find the sum of the digits of a given number using recursive function

#finding the sum of the digits of a number using function

```
def sod(n):  
    if(n==0):  
        return 0  
    else:  
        return (n%10+sod(n//10))
```

```
n=int(input("enter a positive number"))
r=sod(n)
print("sum of the digits of",n,"is",r)
```

3. Write a Program to find the nth term of a Fibonacci series using recursive function

```
def fibonacci(n):
    if(n==1):
        return 0
    elif(n==2):
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

```
n=int(input("enter a number"))
r=fibonacci(n)
print(n,"Fibonacci number is",r)
```

4. Write a Program to find the reverse of a given number using recursive function or recursion

```
rev=0
def reverse(n):
    if(n==0):
        return 0
    else:
        d=n%10
        rev=rev*10+d
        n=n//10
        reverse(n)
    return rev
```

```
n=int(input("enter a number"))
r=reverse(n)
```

```
print("reverse number of",n,"is",r)
```

What is the output of the following code

```
1. def fun(x,y):
    if y==0:
        return 1
    else:
        return x*fun(x,y-1)
print(fun(2,3))
```

Answer: The above code is recursive function for finding **x^y**

So result is $2^3 = 8$

2. What is the output of the following code

```
def fun(x,y):
    if y==0:
        return 0
    else:
        return x+fun(x,y-1)
print(fun(2,3))
```

Answer: The above code is recursive function for finding **$x * y$**

So result is $2*3=6$

Important Questions from Functions:

1. Define a function. What are the advantages of the functions.
2. A) Write the syntax for defining a function
b) Define calling function, called function, formal arguments and actual arguments
3. Explain various types of arguments.
4. Define a recursion or recursive function and write program to find the factorial of a number using recursion or recursive function.
5. A) Write a program to find n^{th} term of a Fibonacci series using recursion or recursive function.
b) Explain local and global variables briefly
- 6 a) Write program to find the factorial of a number using function.
b) Write program to find the sum of the digits of a given number using recursive function.

FILES

Definition:

A file is a collection of data stored on a secondary storage device like hard disk.

A file is basically used because real-life applications involve large amounts of data and in such situations the **console oriented I/O (input() ,print() functions from python)** operations pose **two major problems**:

- **First**, it becomes time consuming to handle huge amount of data through terminals.
- **Second**, when doing I/O using terminal, the entire data is lost when either the program is terminated or computer is turned off.

Therefore, it becomes necessary to store data on a permanent storage (the disks) and read whenever necessary, without destroying the data.

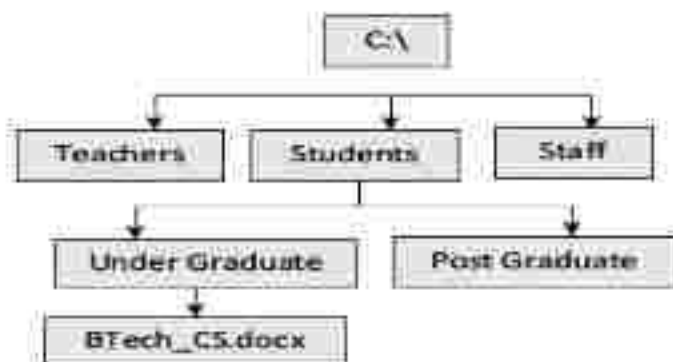
File Path:

It denotes the location of the file.

Relative Path and Absolute Path

A file path can be either *relative* or *absolute*. While an **absolute path** always contains the root and the complete directory list to specify the exact location the file, **a relative path** as only a part of the complete path is specified.

Example:



C:\Students\Under Graduate\BTech_CS.docx is an absolute path.

but

Under Graduate\BTech_CS.docx is a relative path

Types of Files:

Based on content or data of a file. Files are two types

- Text files
- Binary files

Text files:

Text files contain data in the form of letters, digits or special symbols which can be read and understood by humans.

Ex: abc.txt, c7pl.c, p.py

Binary files:

Binary files contain data in the form of bits that can be easily understood by computers and not by humans.

Ex: all audio, video, images, 1.exe, xyz.dll // exe means executable files

To make use of files, we have to follow the steps

First we have to open the file, second do some kind of operation on file and then close the file.

1. The Open() Function:

Before reading from or writing to a file, you must first open it using Python's built-in open() function. This function creates a file object, which will be used to invoke methods associated with it. The syntax of open() is:

```
fileObj = open("file_name", "access_mode")
```

Here,

file_name is a string value that specifies name of the file that you want to access.

access_mode indicates the mode in which the file has to be opened (purpose of opening the file), i.e., read, write, append, etc.

Example:

```
file = open("File1.txt", "rb")  
print(file)
```

OUTPUT

```
<open file 'File1.txt', mode 'rb' at 0x02A850D0>
```

The open() Function – Access Modes or modes:

Text files modes:

S.no	Mode	Symbol	Purpose	Example
1	Write	"w"	It opens a file for writing purpose. If already the file exists then the file contents will be overwritten otherwise a new file will be created.	P=open("x.txt","w")
2	Read	"r"	It opens a file for reading purpose. If already the file exists then the file contents will be read one by one by the file pointer or object otherwise error .	P=open("x.txt","r")
3	Append	"a"	It opens a file for appending purpose. If already the file exists then the file contents will be added at the	P=open("x.txt","a")

			end of the file otherwise a newfile will be created.	
Text files mixed modes:				
1	Write+read	"w+"	It opens a file for writing and reading purpose. If already the file exists then the file contents will be overwritten otherwise a new file will be created.	P=open("x.txt","w+")
2	Append+read	"a+"	It opens a file for appending and reading purpose. If already the file exists then the file contents will be added at the end of the file otherwise a new file will be created.	P=fopen("x.txt","a+")
3	Read+write	"r+"	It opens a file for reading and writing purpose. If already the file exists then the file contents will be read one by one by the file pointer	P=open("x.txt","r+")

			otherwise Error	
--	--	--	------------------------	--

Binary File modes:

S.no	Mode	Symbol	Purpose	Example
1	Write	"wb"	It opens a file for writing purpose. If already the file exists then the file contents will be overwritten otherwise a new file will be created.	P=open("x.txt","wb")
2	Read	"rb"	It opens a file for reading purpose. If already the file exists then the file contents will be read one by one by the file pointer otherwise Error	P=open("x.txt","rb")
3	Append	"ab"	It opens a file for appending purpose. If already the file exists then the file contents will be added at the end of the file otherwise a new file will be created.	P=open("x.txt","ab")

Binary files mixed modes:

1	Write+read	"w+b"	It opens a file for	P=open("x.txt","w+b")
---	------------	-------	---------------------	-----------------------

			writing and reading purpose. If already the file exists then the file contents will be overwritten otherwise a new file will be created.	
2	Append+read	"a+b"	It opens a file for appending and reading purpose. If already the file exists then the file contents will be added at the end of the file otherwise a new file will be created.	P=fopen("x.txt","a+b");
3	Read+write	"r+b"	It opens a file for reading and writing purpose. If already the file exists then the file contents will be read one by one by the file pointer otherwise Error	P=open("x.txt","r+b")

The File Object Attributes

Once a file is successfully opened, a *file* object is returned. Using this file object, you can easily access different type of information related to that file. This information can be obtained by reading values of specific attributes of the file.

Attribute	Information Obtained
<code>fileObj.closed</code>	Returns true if the file is closed and false otherwise
<code>fileObj.mode</code>	Returns access mode with which file has been opened
<code>fileObj.name</code>	Returns name of the file

Example:

```
file = open("File1.txt", "wb")
print("Name of the file: ", file.name)
print("File is closed.", file.closed)

print("File has been opened in ", file.mode, "mode")
```

OUTPUT

```
Name of the file: File1.txt
File is closed. False
File has been opened in wb mode
```

The close () Method

The *close()* method is used to close the file object. Once a file object is closed, you cannot further read from or write into the file associated with the file object .

The syntax of close() is

```
fileObj.close()
```

Reason for closing the file:

The *close()* method frees up any system resources such as file descriptors, file locks, etc. that are associated with the file. Moreover, there is an upper limit to the number of files a program can open. If that limit is exceeded then the program may even crash or work in

unexpected manner. Thus, you can waste lots of memory if you keep many files open unnecessarily and also remember that open files always stand a chance of corruption and data loss.

The write() and writelines() Methods

The *write()* method is used to write a string to an already opened file. Of course this string may include numbers, special characters or other symbols.

While writing data to a file, you must remember that the *write()* method does not add a newline character ('\n') to the end of the string.

The syntax of write() method is: fileObj.write(string)

The writelines() method is used to write a list of strings.

Examples:

```
file = open("File1.txt", "w")
file.write("Hello All, hope you are enjoying learning Python")
file.close()
print("Data Written into the file.....")
```

OUTPUT

```
Data Written into the file....."
```

Example on Writeline ()method:

```
file = open("File1.txt", "w")
lines = ["Hello World, ", "Welcome to the world of Python", "Enjoy learning Python"]
file.writelines(lines)
file.close()
print("Data written to file.....")
```

OUTPUT

```
Data written to file.....
```

How to append the data?

Once you have stored some data in a file, you can always open that file again to write more data or append data to it. To append a file, you must

open it using 'a' or 'ab' mode depending on whether it is a text file or a binary file.

Example:

```
file = open("File1.txt", "a")
file.write("\n Python is a very simple yet powerful language")
file.close()
print("Data appended to file.....")
```

OUTPUT

Data appended to file....."

The read() and readline() Methods

The **read() method** is used to read a string from an already opened file. As said before, the string can include, alphabets, numbers, characters or other symbols.

The syntax of read() method is **fileObj.read([count])**

In the above syntax, count is an optional parameter which if passed to the read() method specifies the number of bytes to be read from the opened file.

The read() method starts reading from the beginning of the file and *if count is missing or has a negative value then, it reads the entire contents of the file (i.e., till the end of file).*

Example:

```
file = open("File1.txt", "r")
print(file.read(10))
file.close()
```

OUTPUT

Hello All,

The readlines() method is used to read all the lines in the file.

Example:

```
f=open("1.txt","r")
```



```
for line in f.readlines():  
    print(line)
```

Splitting Words

Python allows you to read line(s) from a file and splits the line (treated as a string) based on a character. By default, this character is space but you can even specify any other character to split words in the string.

Example:

```
with open('file1.txt', 'r') as file:  
    line = file.readline()  
    words = line.split()  
    print(words)
```

OUTPUT

```
['Hello', 'world', ' ', 'Welcome', 'to', 'the', 'world', 'of', 'Python', 'Programming']
```

Programs:

1. Write a python program to read contents from a file and display the contents

Source code:

```
f=open("python.txt","w")  
f.write("this is my first file program")  
f.close()  
f=open("python.txt","r")  
print(f.read())  
f.close()
```

2. Write a python program to display the number of characters, digits and special characters present in the given file content

Source code:

```
s=0  
d=0  
a=0  
f=open("p26.py","r")  
t=f.read()  
for c in t:
```

```

    if(c.isdigit()):
        d=d+1
    elif(c.isalpha()):
        a=a+1
    else:
        s=s+1
print("number of alphabets in a file",a)
print("number of digits in a file",d)
print("number of special characters in a file",s)
f.close()

```

3.

You are given a file called grades.txt, where each line of the file contains a one-word student username and three test scores separated by spaces, like below:.

Rathan 83 77 54

Adams 86 69 90

Write code that scans through the file and determines how many students passed all three tests (**Hint:- Pass % based on user input**).

Source code:

```

f=open('listfile.txt', 'r')
l=[]
c=0
for line in f:
    l.append(line)
    l[c]=line
    l[c]=l[c].split( )
    c=c+1
for i in range(0,c):
    c1=0
    print(l[i])
    for j in range(1,4):
        if(int(l[i][j])>35):
            c1=c1+1
    if(c1==3):
        print("pass")

```

```
else:  
    print("fail")
```

```
f.close()
```

4. Write a Program to copy the contents of one file into another file.

```
f=open("1.py","r") #f is referring first file to read  
p=open("2.py","w") # p is referring second file to write  
p.write(f.read()) # we read contents from file f and write in another  
#file called p  
f.close()  
p.close()
```

5. Write a Program to merge the contents of two files into third file.

```
f=open("1.py","r") #f is referring first file to read  
p=open("2.py","r") # p is referring second file to read  
q=open("mergedfile.py","w") # q is referring merged file to write  
q.write(f.read()) #copy from first file to merged file.  
f.close() #close first file  
q=open("mergedfile.py","a") #open merged file in append mode  
q.write(p.read()) #copy from second file to merged file.  
p.close() #close first file  
q.close() #close first file
```